Microsoft

MCTS EXAM
70-502

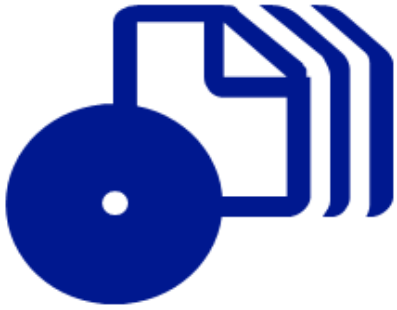
Microsoft® .NET Framework 3.5– Windows® Presentation Foundation

Matthew A. Stoecker

SELF-PACED

Training Kit

How to access your CD files



The print edition of this book includes a CD. To access the CD files, go to http://aka.ms/625662/files, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008929780

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWE 3 2 1 0 9 8

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to tkinput@microsoft.com.

Acquisitions Editor: Ken Jones
Developmental Editor: Laura Sackerman
Project Editor: Kathleen Atkins
Editorial Production: S4Carlisle Publishing Services
Technical Reviewer: Kurt Meyer; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X14-15151

About the Author

Matthew A. Stoecker

Matthew Stoecker started programming in BASIC on a TRS-80 at the age of nine. In 2001, he joined Microsoft Corporation as a programming writer authoring documentation for Microsoft Visual Basic .NET. He has written numerous technical articles about Visual Basic .NET and Visual C#, and he has written or contributed to multiple books about these languages, Windows Forms, and now Windows Presentation Foundation (WPF). He holds a Bachelor of Music degree in trombone performance from the Oberlin Conservatory and a Ph.D in microbiology from the University of Washington that he hopes he will never have to use again. He spends his spare time biking, playing the trombone, and playing with his cats. He lives in Bellevue, Washington.

Contents at a Glance

1 WPF Application Fundamentals 1
2 Events, Commands, and Settings 57
3 Building the User Interface 99
4 Adding and Managing Content 153
5 Configuring Databinding 207
6 Converting and Validating Data 259
7 Styles and Animation 303
8 Customizing the User Interface 343
9 Resources, Documents, and Localization 389
10 Deployment 441

Answers 473

Glossary 499

Index 503

Table of Contents

Introduction . . . . . **xxi**

1 WPF Application Fundamentals **1**

Before You Begin 2

Lesson 1: Selecting an Application Type 3

Application Type Overview 3

Windows Applications 4

Navigation Applications 9

XBAPs 11

Security and WPF Applications 13

Choosing an Application Type 14

Lab: Creating WPF Applications 15

Lesson Summary 19

Lesson Review 19

Lesson 2: Configuring Page-Based Navigation 21

Using Pages 21

Hosting Pages in Frames 21

Using Hyperlinks 22

Using *NavigationService* 23

Using the Journal 25

Handling Navigation Events 27

Using *PageFunction* Objects 30

Simple Navigation and Structured Navigation 32

Lab: The Pizza Kitchen 32

Lesson Summary 38

Lesson Review 39

Lesson 3: Managing Application Responsiveness 41
Running a Background Process 42
Providing Parameters to the Process 43
Returning a Value from a Background Process 44
Cancelling a Background Process 45
Reporting the Progress of a Background Process with *BackgroundWorker* 46
Using *Dispatcher* to Access Controls Safely on Another Thread 47
Freezable Objects 48
Lab: Practicing with *BackgroundWorker* 49
Lesson Summary 51
Lesson Review 52
Chapter Review 53
Chapter Summary 53
Key Terms 53
Case Scenario 54
Case Scenario: Designing a Demonstration Program 54
Suggested Practices 55
Take a Practice Test 55

2 Events, Commands, and Settings 57

Before You Begin 57
Lesson 1: Configuring Events and Event Handling 59
RoutedEventArgs 61
Attaching an Event Handler 62
The *EventManager* Class 63
Defining a New Routed Event 64
Creating a Class-Level Event Handler 66
Application-Level Events 66
Lab: Practice with Routed Events 68
Lesson Summary 69
Lesson Review 70
Lesson 2: Configuring Commands 72
A High-Level Procedure for Implementing a Command 73
Invoking Commands 74

Command Handlers and Command Bindings 75
Creating Custom Commands 78
Lab: Creating a Custom Command 80
Lesson Summary 83
Lesson Review 84
Lesson 3: Configuring Application Settings 86
Creating Settings at Design Time 87
Loading Settings at Run Time 88
Saving User Settings at Run Time 88
Lab: Practice with Settings 89
Lesson Summary 91
Lesson Review 91
Chapter Review 94
Chapter Summary 94
Key Terms 95
Case Scenarios 95
Case Scenario 1: Validating User Input 95
Case Scenario 2: Humongous Insurance User Interface 96
Suggested Practices 96
Take a Practice Test 97

3 Building the User Interface 99

Before You Begin 99
Lesson 1: Using Content Controls 101
WPF Controls Overview 101
Content Controls 101
Other Controls 105
Using Attached Properties 110
Setting the Tab Order for Controls 111
Lab: Building a User Interface 111
Lesson Summary 113
Lesson Review 113
Lesson 2: Item Controls 116
ListBox Control 116
ComboBox Control 117

TreeView Control . . . 118
Menus . . . 119
ToolBar Control . . . 121
StatusBar Control . . . 123
Virtualization in Item Controls . . . 123
Lab: Practice with Item Controls . . . 124
Lesson Summary . . . 127
Lesson Review . . . 127
Lesson 3: Using Layout Controls . . . 130
Control Layout Properties . . . 130
Layout Panels . . . 132
Accessing Child Elements Programmatically . . . 143
Aligning Content . . . 144
Lab: Practice with Layout Controls . . . 146
Lesson Summary . . . 148
Lesson Review . . . 149
Chapter Review . . . 150
Chapter Summary . . . 150
Key Terms . . . 150
Case Scenarios . . . 151
Case Scenario 1: Streaming Stock Quotes . . . 151
Case Scenario 2: The Stock Watcher . . . 151
Suggested Practices . . . 152
Take a Practice Test . . . 152

4 Adding and Managing Content . . . 153

Before You Begin . . . 153
Lesson 1: Creating and Displaying Graphics . . . 155
Brushes . . . 155
Shapes . . . 163
Transformations . . . 168
Clipping . . . 171
Hit Testing . . . 171
Lab: Practice with Graphics . . . 172

Lesson Summary . . . 173
Lesson Review . . . 174
Lesson 2: Adding Multimedia Content . . . 176
Using *SoundPlayer* . . . 176
MediaPlayer and *MediaElement* . . . 179
Handling Media-Specific Events . . . 182
Lab: Creating a Basic Media Player . . . 183
Lesson Summary . . . 185
Lesson Review . . . 185
Lesson 3: Managing Binary Resources . . . 187
Embedding Resources . . . 187
Loading Resources . . . 188
Retrieving Resources Manually . . . 189
Content Files . . . 190
Retrieving Loose Files with *siteOfOrigin* Pack URIs . . . 190
Lab: Using Embedded Resources . . . 191
Lesson Summary . . . 192
Lesson Review . . . 192
Lesson 4: Managing Images . . . 194
The *Image* Element . . . 194
Stretching and Sizing Images . . . 194
Transforming Graphics into Images . . . 196
Accessing Bitmap Metadata . . . 198
Lab: Practice with Images . . . 200
Lesson Summary . . . 201
Lesson Review . . . 202
Chapter Review . . . 204
Chapter Summary . . . 204
Key Terms . . . 204
Case Scenarios . . . 205
Case Scenario 1: The Company with Questionable Taste . . . 205
Case Scenario 2: The Image Reception Desk . . . 205
Suggested Practices . . . 206
Take a Practice Test . . . 206

5 Configuring Databinding 207
Before You Begin 208
Lesson 1: Configuring Databinding 209
The *Binding* Class 209
Binding to a WPF Element 211
Binding to an Object 212
Setting the Binding Mode 215
Setting the *UpdateSourceTrigger* Property 216
Lab: Practice with Bindings 217
Lesson Summary 218
Lesson Review 219
Lesson 2: Binding to Data Sources 221
Binding to a List 221
Binding an Item Control to a List 221
Binding a Single Property to a List 223
Navigating a Collection or List 223
Binding to ADO.NET Objects 226
Setting the *DataContext* to an ADO.NET *DataTable* 226
Setting the *DataContext* to an ADO.NET *DataSet* 227
Binding to Hierarchical Data 228
Binding to Related ADO.NET Tables 228
Binding to an Object with *ObjectDataProvider* 230
Binding to XML Using the *XmlDataProvider* 231
Using *XPath* with *XmlDataProvider* 232
Lab: Accessing a Database 232
Lesson Summary 235
Lesson Review 236
Lesson 3: Manipulating and Displaying Data 238
Data Templates 238
Setting the Data Template 240
Sorting Data 241
Applying Custom Sorting 242
Grouping 243
Creating Custom Grouping 245

Filtering Data 246
Filtering ADO.NET Objects 247
Lab: Practice with Data Templates and Groups 248
Lesson Summary 252
Lesson Review 252
Chapter Review 255
Chapter Summary 255
Key Terms 256
Case Scenarios 256
Case Scenario 1: Getting Information from the Field 256
Case Scenario 2: Viewing Customer Data 257
Suggested Practices 257
Take a Practice Test 258

6 Converting and Validating Data 259

Before You Begin 259
Lesson 1: Converting Data 261
Implementing *IValueConverter* 261
Using Converters to Format Strings 264
Using Converters to Return Objects 268
Using Converters to Apply Conditional Formatting in Data Templates 269
Localizing Data with Converters 271
Using Multi-value Converters 273
Lab: Applying String Formatting and Conditional Formatting 276
Lesson Summary 279
Lesson Review 279
Lesson 2: Validating Data and Configuring Change Notification 282
Validating Data 282
Binding Validation Rules 282
Setting *ExceptionValidationRule* 283
Implementing Custom Validation Rules 283
Handling Validation Errors 284
Configuring Data Change Notification 287
Implementing *INotifyPropertyChanged* 287

Using *ObservableCollection* 288
Lab: Configuring Change Notification and Data Validation 289
Lesson Summary 294
Lesson Review 295
Chapter Review 300
Chapter Summary 300
Key Terms 300
Case Scenarios 301
Case Scenario 1: The Currency Trading Review Console 301
Case Scenario 2: Currency Trading Console 301
Suggested Practices 302
Take a Practice Test 302

7 Styles and Animation 303

Before You Begin 303
Lesson 1: Styles 305
Using Styles 305
Properties of Styles 305
Setters 306
Creating a Style 308
Implementing Style Inheritance 311
Triggers 312
Property Triggers 313
Multi-triggers 314
Data Triggers and Multi-data-triggers 315
Event Triggers 315
Understanding Property Value Precedence 316
Lab: Creating High-Contrast Styles 318
Lesson Summary 320
Lesson Review 320
Lesson 2: Animations 323
Using Animations 323
Important Properties of Animations 324
Storyboard Objects 326

Using *Animations* with *Triggers* 327
Managing the Playback Timeline 330
Animating Non-Double Types 332
Creating and Starting Animations in Code 335
Lab: Improving Readability with Animations 336
Lesson Summary 337
Lesson Review 338
Chapter Review 339
Chapter Summary 339
Key Terms 340
Case Scenarios 340
Case Scenario 1: Cup Fever 340
Case Scenario 2: A Far-Out User Interface 341
Suggested Practices 341
Take a Practice Test 342

8 Customizing the User Interface 343

Before You Begin 343
Lesson 1: Integrating Windows Forms Controls 345
Using Windows Forms Controls 345
Using Dialog Boxes in WPF Applications 345
WindowsFormsHost 349
Using *MaskedTextBox* in WPF Applications 351
Using the *PropertyGrid* in WPF Applications 353
Lab: Practice with Windows Forms Elements 354
Lesson Summary 356
Lesson Review 357
Lesson 2: Using Control Templates 359
Using Control Templates 359
Creating Control Templates 359
Inserting a *Trigger* in a Template 362
Respecting the Templated Parent's Properties 363
Applying Templates with a *Style* 365
Viewing the Source Code for an Existing Template 365

Using Predefined Part Names in a Template . . . 366
Lab: Creating a Control Template . . . 367
Lesson Summary . . . 369
Lesson Review . . . 369
Lesson 3: Creating Custom and User Controls . . . 372
Control Creation in WPF . . . 372
Choosing Among User Controls, Custom Controls, and Templates . . . 373
Implementing and Registering Dependency Properties . . . 373
Creating User Controls . . . 376
Creating Custom Controls . . . 376
Consuming User Controls and Custom Controls . . . 377
Rendering a Theme-Based Appearance . . . 378
Lab: Creating a Custom Control . . . 380
Lesson Summary . . . 383
Lesson Review . . . 383
Chapter Review . . . 385
Chapter Summary . . . 385
Key Terms . . . 385
Case Scenarios . . . 386
Case Scenario 1: Full Support for *Styles* . . . 386
Case Scenario 2: The Pizza Progress Bar . . . 386
Suggested Practices . . . 387
Take a Practice Test . . . 387

9 Resources, Documents, and Localization . . . 389

Before You Begin . . . 389
Lesson 1: Logical Resources . . . 391
Using Logical Resources . . . 391
Logical Resources . . . 392
Creating a Resource Dictionary . . . 395
Retrieving Resources in Code . . . 396
Lab: Practice with Resources . . . 397
Lesson Summary . . . 399
Lesson Review . . . 399

Lesson 2: Using Documents in WPF 401
Flow Documents 401
Creating Flow Documents 402
XPS Documents 418
Viewing XPS Documents 418
Printing 418
Printing Documents 419
The *PrintDialog* Class 419
Lab: Creating a Simple Flow Document 421
Lesson Summary 422
Lesson Review 423
Lesson 3: Localizing a WPF Application 426
Localization 426
Localizing an Application 427
Using Culture Settings in Validators and Converters 432
Lab: Localizing an Application 433
Lesson Summary 436
Lesson Review 436
Chapter Review 438
Chapter Summary 438
Key Terms 438
Case Scenario 439
Case Scenario: Help for the Beta 439
Suggested Practices 440
Take a Practice Test 440

10 Deployment 441

Before You Begin 441
Lesson 1: Creating a Setup Project with Windows Installer 443
Deploying a WPF Application 443
Choosing Between Windows Installer and ClickOnce 443
Deploying with Windows Installer 444
Deploying a Stand-alone Application 445
Creating the Setup Project 445

Adding Files to the Setup Project with the File System Editor 445
Other Setup Project Editors 448
Lab: Creating a Setup Project 448
Lesson Summary 450
Lesson Review 450
Lesson 2: Deploying Your Application with ClickOnce 451
Deploying with ClickOnce 451
Deploying an Application Using ClickOnce 452
Configuring ClickOnce Update Options 455
Deploying an XBAP with ClickOnce 458
Configuring the Application Manifest 461
Associating a Certificate with the Application 463
Lab: Publishing Your Application with ClickOnce 464
Lesson Summary 465
Lesson Review 465
Chapter Review 469
Chapter Summary 469
Key Terms 469
Case Scenario 470
Case Scenario: Buggy Beta 470
Suggested Practices 470
Take a Practice Test 471

Answers 473

Glossary 499

Index 503

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Acknowledgments

Thank you to my friends and family. It isn't easy dealing with a person who is going crazy trying to write a book. Thanks for understanding when I had to work all night. Thanks for understanding when I needed to stay home and write on Friday night instead of going to see a movie. Thanks for putting up with me when all I talked about was how I needed to work on this book. This especially means thanks to you, Libby.

Introduction

This training kit is designed for developers who plan to take the Microsoft Certified IT Professional (MCITP) Exam 70-502, as well as for developers who need to know how to develop Microsoft Windows Presentation Foundation (WPF)–based applications using Microsoft .NET Framework 3.5. We assume that before using this training kit, you already have a working knowledge of Windows, Microsoft Visual Basic or C# (or both), and Extensible Application Markup Language (XAML).

By using this training kit, you will learn how to do the following:

- Create a WPF application
- Build user interfaces using WPF controls
- Add and manage content in a WPF application
- Bind WPF controls to data sources
- Customize the appearance of your WPF application
- Configure a WPF application
- Deploy a WPF application to its intended audience

Hardware Requirements

The following hardware is required to complete the practice exercises:

- A computer with a 1.6-gigahertz (GHz) or faster processor
- A minimum of 384 megabytes (MB) of random access memory (RAM)
- A minimum of 2.2 gigabytes (GB) of available hard disk space is required to install VS 2008. Additionally, 50 megabytes (MB) of available hard disk space is required to install the labs.
- A DVD-ROM drive
- A 1024 x 768 or higher resolution display with 256 colors or more
- A keyboard and Microsoft mouse or compatible pointing device

Software Requirements

The following software is required to complete the practice exercises:

- One of the following operating systems:
  - Windows Vista (any edition except Windows Vista Starter)
  - Windows XP with Service Pack 2 or later (any edition except Windows XP Starter)
  - Windows Server 2003 with Service Pack 1 or later (any edition)
  - Windows Server 2003 R2 or later (any edition)
  - Windows Server 2008
- Microsoft Visual Studio 2008

NOTE A 90-day evaluation edition of Visual Studio 2008 Professional Edition is included on a DVD that comes with this training kit.

Using the CD and DVD

A companion CD and an evaluation software DVD are included with this training kit. The companion CD contains the following:

- **Practice Tests** You can reinforce your understanding of how to create WPF applications in Visual Studio 2008 with .NET Framework 3.5 by using electronic practice tests that you can customize to meet your needs from the pool of Lesson Review questions in this book. Alternatively, you can practice for the 70-502 certification exam by using tests created from a pool of 200 realistic exam questions, which will give you enough different practice tests to ensure you're prepared.
- **Sample Files** Most chapters in this training kit include sample files that are associated with the lab exercises at the end of every lesson. For some exercises, you are instructed to open a project prior to starting the exercise. For other exercises, you create a project on your own and can reference a completed project on the CD if you have a problem following the exercise procedures. Sample files can be installed to your hard drive by simply copying them to the appropriate directory.

After copying the sample files from the CD to your hard drive you must clear the Read Only attribute in order to work with the files on your hard drive.

- **eBook** An electronic version (eBook) of this training kit is included for use at times when you don't want to carry the printed book with you. The eBook is in Portable Document Format (PDF) and you can view it by using Adobe Acrobat or Adobe Reader. You can use the eBook to cut and paste code as you work through the exercises.

The evaluation software DVD contains a 90-day evaluation edition of Visual Studio 2008 Professional Edition, in case you want to use it instead of a full version of Visual Studio 2008 to complete the exercises in this book.

Digital Content for Digital Book Readers: If you bought a digital-only edition of this book, you can enjoy select content from the print edition's companion CD.
Visit *http://www.microsoftpressstore.com/title/9780735625662* to get your downloadable content. This content is always up-to-date and available to all readers.

How to Install the Practice Tests

To install the practice test software from the companion CD to your hard disk, perform the following steps:

1. Insert the companion CD into your CD drive and accept the license agreement that appears onscreen. A CD menu appears.

NOTE If the CD menu or the license agreement doesn't appear, AutoRun might be disabled on your computer. Refer to the Readme.txt file on the CD-ROM for alternative installation instructions.

2. Click Practice Tests and follow the instructions on the screen.

How to Use the Practice Tests

To start the practice test software, follow these steps:

1. Click Start and select All Programs and Microsoft Press Training Kit Exam Prep. A window appears that shows all the Microsoft Press training kit exam prep suites that are installed on your computer.
2. Double-click the lesson review or practice test you want to use.

Lesson Review Options

When you start a lesson review, the Custom Mode dialog box appears, allowing you to configure your test. You can click OK to accept the defaults or you can customize the number of questions you want, the way the practice test software works, which exam objectives you want the questions to relate to, and whether you want your lesson review to be timed. If you are retaking a test, you can select whether you want to see all the questions again or only those questions you previously skipped or answered incorrectly.

After you click OK, your lesson review starts. You can take the test as follows:

- To take the test, answer the questions and use the Next, Previous, and Go To buttons to move from question to question.
- After you answer an individual question, if you want to see which answers are correct, along with an explanation of each correct answer, click Explanation.
- If you would rather wait until the end of the test to see how you did, answer all the questions and then click Score Test. You see a summary of the exam objectives that you chose and the percentage of questions you got right overall and per objective. You can print a copy of your test, review your answers, or retake the test.

Practice Test Options

When you start a practice test, you can choose whether to take the test in Certification Mode, Study Mode, or Custom Mode.

- **Certification Mode** Closely resembles the experience of taking a certification exam. The test has a set number of questions, it is timed, and you cannot pause and restart the timer.
- **Study Mode** Creates an untimed test in which you can review the correct answers and the explanations after you answer each question.
- **Custom Mode** Gives you full control over the test options so that you can customize them as you like.

In all modes, the user interface you see when taking the test is basically the same, but different options are enabled or disabled, depending on the mode. The main options are discussed in the previous section, "Lesson Review Options."

When you review your answer to an individual practice test question, a "References" section is provided. This section lists where in the training kit you can find the information

that relates to that question, and it also provides links to other sources of information. After you click Test Results to score your entire practice test, you can click the Learning Plan tab to see a list of references for every objective.

How to Uninstall the Practice Tests

To uninstall the practice test software for a training kit, use the Add Or Remove Programs option in the Control Panel in Windows.

Microsoft Certified Professional Program

Microsoft certifications provide the best method to prove your command of current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop or implement and support solutions with Microsoft products and technologies. Computer professionals who become Microsoft certified are recognized as experts and are sought after industry wide. Certification brings a variety of benefits to the individual and to employers and organizations.

MORE INFO For a full list of Microsoft certifications, go to *http://www.microsoft.com/learning/mcp/default.mspx*.

Technical Support

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD. If you have comments, questions, or ideas regarding this book or the companion CD, please send them to Microsoft Press by using either of the following methods:

E-mail: tkinput@microsoft.com

Postal Mail:

Microsoft Press

Attn: *MCITP Self-Paced Training Kit (Exam 70-502) Microsoft .NET Framework 3.5 – Windows Presentation Foundation* Editor

One Microsoft Way

Redmond, WA, 98052-6399

For additional support information regarding this book and the CD-ROM (including answers to commonly asked questions about installation and use), visit the Microsoft Press Technical Support Web site at *http://www.microsoft.com/learning/support/books*. To connect directly to the Microsoft Knowledge Base and enter a query, visit *http://support.microsoft.com/search*. For support information regarding Microsoft software, please connect to *http://support.microsoft.com*.

Evaluation Edition Software

The 90-day evaluation edition provided with this training kit is not the full retail product and is provided only for the purposes of training and evaluation. Microsoft and Microsoft Technical Support do not support this evaluation edition.

Information about any issues relating to the use of this evaluation edition with this training kit is posted in the Support section of the Microsoft Press Web site (*http://www.microsoft.com/learning/support/books/*). For information about ordering the full version of any Microsoft software, please call Microsoft Sales at (800) 426-9400 or visit *http://www.microsoft.com*.

Chapter 2

Events, Commands, and Settings

Events and commands form the basis of the architecture for intra-application communication in Windows Presentation Foundation (WPF) applications. Routed events can be raised by multiple controls and allow a fine level of control over user input. Commands are a welcome addition to the Microsoft .NET Framework and provide a central architecture for enabling and disabling high-level tasks. Application settings allow you to persist values between application sessions. In this chapter, you will learn to configure these features.

Exam objectives in this chapter:

- Configure event handling.
- Configure commands.
- Configure application settings.

Lessons in this chapter:

- Lesson 1: Configuring Events and Event Handling 59
- Lesson 2: Configuring Commands 72
- Lesson 3: Configuring Application Settings 86

Before You Begin

To complete the lessons in this chapter, you must have

- A computer that meets or exceeds the minimum hardware requirements listed in the "About This Book" section at the beginning of the book
- Microsoft Visual Studio 2005 Professional Edition installed on your computer
- An understanding of Microsoft Visual Basic or C# syntax and familiarity with the .NET Framework

Real World

Matthew Stoecker

By using WPF routed events and commands, I find I have a much finer control over how my user interfaces respond compared to in a Windows Forms application. The Routed Event architecture allows me to implement complex event handling strategies, and the Command architecture provides a way to approach programming common tasks in my user interfaces.

Lesson 1: Configuring Events and Event Handling

Events in WPF programming are considerably different from those in traditional Windows Forms programming. WPF introduces routed events, which can be raised by multiple controls and handled by multiple handlers. *Routed events* allow you to add multiple levels of complexity and sophistication to your user interface and the way it responds to user input. In this lesson, you will learn about routed events, including how to handle a routed event, define and register a new routed event, handle an application lifetime event, and use the *EventManager* class.

After this lesson, you will be able to:

- Explain the difference between a direct event, a bubbling event, and a tunneling event
- Define and register a new routed event
- Define static class event handlers
- Handle an event in a WPF application
- Handle an attached event in a WPF application
- Handle application lifetime events
- Use the *EventManager* class

Estimated lesson time: 30 minutes

Events have been a familiar part of Microsoft Windows programming for years. An *event* is a message sent by an object, such as a control or other part of the user interface, that the program responds to (or handles) by executing code. While the traditional .NET event architecture is still present in WPF programming, WPF builds upon the event concept by introducing routed events.

A key concept to remember in event routing is the control containment hierarchy. In WPF user interfaces, controls frequently contain other controls. For example, a typical user interface might consist of a top-level *Window* object, which contains a *Grid* object, which itself might contain several controls, one of which could be a *ToolBar* control, which in turn contains several *Button* controls. The routed event architecture allows for an event that originates in one control to be raised by another control in the containment hierarchy. Thus, if the user clicks one of the *Button* controls on the toolbar, that event can be raised by the *Button,* the *ToolBar,* the *Grid,* or the *Window.*

Why is it useful to route events? Suppose, for example, that you are designing a user interface for a calculator program. As part of this application, you might have several

Button controls enclosed within a *Grid* control. Suppose that you wanted all button clicks in this grid to be handled by a single event handler? WPF raises the click event from the *Button,* the *Grid,* and any other control in the control containment hierarchy. As the developer, you can decide where and how the event is handled. Thus, you can provide a single event handler for all *Button Click* events originating from *Button* controls in the grid, simplifying code-writing tasks and ensuring consistency in event handling.

Types of Routed Events There are three different types of routed events: direct, bubbling, and tunneling.

Direct Events

Direct events are most similar to standard .NET events. Like a standard .NET event, a direct event is raised only by the control in which it originates. Because other controls in the control containment hierarchy do not raise these events, there is no opportunity for any other control to provide handlers for these events. An example of a direct event is the *MouseLeave* event.

Bubbling Events

Bubbling events are events that are raised first in the control where they originate and then are raised by each control in that control's control containment hierarchy, also known as a visual tree. The *MouseDown* event is an example of a bubbling event. Suppose that you have a *Label* contained inside a *FlowPanel* contained inside a *Window.* When the mouse button is pressed over the *Label,* the first control to raise the *MouseDown* event would be the *Label.* Then the *FlowPanel* would raise the *MouseDown* event, and then finally the *Window* itself. You could provide an event handler at any or all stages of the event process.

Tunneling Events

Tunneling events are the opposite of bubbling events. A *tunneling event* is raised first by the topmost container in the visual tree and then down through each successive container until it is finally raised by the element in which it originated. An example of a tunneling event is the *PreviewMouseDown* event. In the previous example, although the event originates with the *Label* control, the first control to raise the *PreviewMouseDown* event is the *Window,* then the *FlowPanel,* and then finally the *Label.* Tunneling events allow you the opportunity to intercept and handle events in the window or container before the event is raised by the specific control. This allows you to filter input, such as keystrokes, at varying levels.

In the .NET Framework, all tunneling events begin with the word "Preview," such as *PreviewKeyDown*, *PreviewMouseDown*, etc., and are typically defined in pairs with a complementary bubbling event. For example, the tunneling event *PreviewKeyDown* is paired with the bubbling event *KeyDown*. The tunneling event always is raised before its corresponding bubbling event, thus allowing an opportunity for higher-level controls in the visual tree to handle the event. Each tunneling event shares its instance of event arguments with its paired bubbling event. This fact is important to remember when handling events, and it will be discussed in greater detail later in this chapter.

RoutedEventArgs

All routed events include an instance of *RoutedEventArgs* (or a class that inherits *RoutedEventArgs*) in their signatures. The *RoutedEventArgs* class contains a wealth of information about the event and its source control. Table 2-1 describes the properties of the *RoutedEventArgs* class.

Table 2-1 *RoutedEventArgs* Properties

Property	Description
Handled	Indicates whether or not this event has been handled. By setting this property to *True*, you can halt further event bubbling or tunneling.
OriginalSource	Gets the object that originally raised the event. For most WPF controls, this will be the same as the object returned by the *Source* property. However, for some controls, such as composite controls, this property will return a different object.
RoutedEvent	Returns the *RoutedEvent* object for the event that was raised. When handling more than one event with the same event handler, you might need to refer to this property to distinguish which event has been raised.
Source	Returns the object that raised the event.

All *EventArgs* for routed events inherit the *RoutedEventArgs* class, but many of them provide additional information. For example, *KeyboardEventArgs* is used in keyboard events and provides information about keystrokes. Likewise, *MouseEventArgs*, used in mouse events, provides information about the state of the mouse when the event took place.

Quick Check

- What are the three kinds of routed events in WPF and how do they differ?

Quick Check Answer

- Routed events in WPF come in three different types: direct, tunneling, and bubbling. A direct event can be raised only by the element in which it originated. A bubbling event is raised first by the element in which it originates and then is raised by each successive container in the visual tree. A tunneling event is raised first by the topmost container in the visual tree and then down through each successive container until it is finally raised by the element in which it originated. Tunneling and bubbling events allow elements of the user interface to respond to events raised by their contained elements.

Attaching an Event Handler

The preferred way to attach an event handler is directly in the Extensible Application Markup Language (XAML) code. You set the event to the name of a method with the appropriate signature for that event. The following example demonstrates setting the event handler for a *Button* control's *Click* event, as shown in bold:

```
<Button Height="23" Margin="132,80,70,0" Name="button1"
   VerticalAlignment="Top" Click="button1_Click">Button</Button>
```

Just like setting a property, you must supply a string value that indicates the name of the method.

Attached Events

It is possible for a control to define a handler for an event that the control cannot itself raise. These incidents are called *attached events*. For example, consider *Button* controls in a *Grid*. The *Button* class defines a *Click* event, but the *Grid* class does not. However,

you still can define a handler for buttons in the grid by attaching the *Click* event of the *Button* control in the XAML code. The following example demonstrates attaching an event handler for a *Button* contained in a *Grid:*

```
<Grid Button.Click="button_Click">
   <Button Height="23" Margin="132,80,70,0" Name="button1"
   VerticalAlignment="Top" >Button</Button>
</Grid>
```

Now every time a button contained in the *Grid* shown here is clicked, the *button_Click* event handler will handle that event.

Handling a Tunneling or Bubbling Event

At times, you might want to halt the further handling of tunneling or bubbling events. For example, you might want to suppress keystroke handling at a particular level in the control hierarchy. You can handle an event and halt any further tunneling or bubbling by setting the *Handled* property of the *RoutedEventArgs* instance to *True*, as shown here:

```
' VB
Private Sub TextBox1_KeyDown(ByVal sender As System.Object, _
   ByVal e As System.Windows.Input.KeyEventArgs)
   e.Handled = True
End Sub

// C#
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
   e.Handled = true;
}
```

Note that tunneling events and their paired bubbling events (such as *PreviewKeyDown* and *KeyDown*) share the same instance of *RoutedEventArgs*. Thus, if you set the *Handled* property to *True* on a tunneling event, its corresponding bubbling event also is considered handled and is suppressed.

The *EventManager* Class

EventManager is a static class that manages the registration of all WPF routed events. Table 2-2 describes the methods of the *EventManager* class.

Table 2-2 *EventManager* Methods

Method	Description
GetRoutedEvents	Returns an array that contains all the routed events that have been registered in this application.
GetRoutedEventsForOwner	Returns an array of all the routed events that have been registered for a specified element in this application.
RegisterClassHandler	Registers a class-level event handler, as discussed in the section "Creating a Class-Level Event Handler," later in this chapter.
RegisterRoutedEvent	Registers an instance-level event handler, as discussed in the next section.

Defining a New Routed Event

You can use the *EventManager* class to define a new routed event for your WPF controls. The following procedure describes how to define a new routed event.

▶ **To define a new routed event**

1. Create a static, read-only definition for the event, as shown in this example:

```
' VB
Public Shared ReadOnly SuperClickEvent As RoutedEvent

// C#
public static readonly RoutedEvent SuperClickEvent;
```

2. Create a wrapper for the routed event that exposes it as a traditional .NET Framework event, as shown in this example:

```
' VB
Public Custom Event SuperClick As RoutedEventHandler
   AddHandler(ByVal value As RoutedEventHandler)
      Me.AddHandler(SuperClickEvent, value)
   End AddHandler

   RemoveHandler(ByVal value As RoutedEventHandler)
      Me.RemoveHandler(SuperClickEvent, value)
   End RemoveHandler
```

```
    RaiseEvent(ByVal sender As Object, _
       ByVal e As System.Windows.RoutedEventArgs)
       Me.RaiseEvent(e)
    End RaiseEvent
End Event

// C#
public event RoutedEventHandler SuperClick
{
    add
    {
       this.AddHandler(SuperClickEvent, value);
    }
    remove
    {
       this.RemoveHandler(SuperClickEvent, value);
    }
}
```

Note that you need to use a different *EventArgs* class than *RoutedEventArgs*. You need to derive a new class from *RoutedEventArgs* and create a new delegate that uses those event arguments.

3. Use *EventManager* to register the new event in the constructor of the class that owns this event. You must provide the name of the event, the routing strategy (direct, tunneling, or bubbling), the type of delegate that handles the event, and the type of the class that owns it. An example is shown here:

```
' VB
EventManager.RegisterRoutedEvent("SuperClick", _
   RoutingStrategy.Bubble, GetType(RoutedEventArgs), GetType(Window1))

// C#
EventManager.RegisterRoutedEvent("SuperClick",
   RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(Window1));
```

Raising an Event

Once an event is defined, you can raise it in code by creating a new instance of *RoutedEventArgs* and using the *RaiseEvent* method, as shown here:

```
' VB
Dim myEventArgs As New RoutedEventArgs(myControl.myNewEvent)
MyBase.RaiseEvent(myEventArgs)

// C#
RoutedEventArgs myEventArgs = new RoutedEventArgs(myControl.myNewEvent);
RaiseEvent(myEventArgs);
```

Creating a Class-Level Event Handler

You can use the *EventManager* class to register a class-level event handler. A class-level event handler handles a particular event for all instances of a class, and is always invoked before instance handlers. Thus, you can screen and suppress events before they reach instance handlers. The following procedure describes how to implement a class-level event handler.

▶ To create a class-level event handler

1. Create a static method to handle the event. This method must have the same signature as the event. An example is shown here:

```
' VB
Private Shared Sub SuperClickHandlerMethod(ByVal sender As Object, _
   ByVal e As RoutedEventArgs)
   ' Handle the event here
End Sub

// C#
private static void SuperClickHandlerMethod(object sender, RoutedEventArgs e)
{
   // Handle the event here
}
```

2. In the static constructor for the class for which you are creating the class-level event handler, create a delegate to this method, as shown here:

```
' VB
Dim SuperClickHandler As New RoutedEventHandler( _
    AddressOf SuperClickHandlerMethod)

// C#
RoutedEventHandler SuperClickHandler = new
   RoutedEventHandler(SuperClickHandlerMethod);
```

3. Also in the static constructor, call *EventManager.RegisterClassHandler* to register the class-level event handler, as shown here:

```
' VB
EventManager.RegisterClassHandler(GetType(Window1), _
   SuperClickEvent, SuperClickHandler)

// C#
EventManager.RegisterClassHandler(typeof(Window1),
   SuperClickEvent,SuperClickHandler);
```

Application-Level Events

Every WPF application is wrapped by an *Application* object. The *Application* object provides a set of events that relate to the application's lifetime. You can handle these events

to execute code in response to application startup or closure. The *Application* object also provides a set of events related to navigation in page-based applications. These events were discussed in Chapter 1, "WPF Application Fundamentals." Table 2-3 describes the available application-level events, excluding the navigation events.

Table 2-3 Selected Application-Level Events

Event	Description
Activated	Occurs when you switch from another application to your program. It also is raised the first time you show a window.
Deactivated	Occurs when you switch to another program.
DispatcherUnhandledException	Raised when an unhandled exception occurs in your application. You can handle an unhandled exception in the event handler for this event by setting the *DispatcherUnhandledException-EventArgs.Handled* property to *True*.
Exit	Occurs when the application is shut down for any reason.
SessionEnding	Occurs when the Windows session is ending, such as when the user shuts down the computer or logs off.
Startup	Occurs as the application is started.

Application events are standard .NET events (rather than routed events), and you can create handlers for these events in the standard .NET way. The following procedure explains how to create an event handler for an application-level event.

▶ To create an application-level event handler

1. In Visual Studio, in the Solution Explorer, right-click Application.xaml (in Visual Basic) or App.xaml (in C#) and choose View Code to open the code file for the *Application* object.
2. Create a method to handle the event, as shown here:

```
' VB
Private Sub App_Startup(ByVal sender As Object, _
   ByVal e As StartupEventArgs)
   ' Handle event here
End Sub
```

```
// C#
void App_Startup(object sender, StartupEventArgs e)
{
   // Handle the event here
}
```

3. In XAML view for the *Application* object, add the event handler to the Application declaration, as shown in bold here:

```
<Application x:Class="Application"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="Window1.xaml" Startup="App_Startup">
```

Lab: Practice with Routed Events

In this lab, you practice using routed events. You create event handlers for the *TextBox.TextChanged* event in three different controls in the visual tree and observe how the event is raised and handled by each one.

Exercise: Creating an Event Handler

1. In Visual Studio, create a new WPF application.
2. From the Toolbox, drag a *TextBox* and three *RadioButton* controls onto the design surface. Note that at this point, these controls are contained by a *Grid* control that is in itself contained in the top-level *Window* control. Thus any bubbling events raised by the *TextBox* will bubble up first to the *Grid* and then to the *Window*.
3. In XAML view, set the display contents of the *RadioButton* controls as follows:

RadioButton	Content
RadioButton1	Handle Textbox.TextChanged in TextBox
RadioButton2	Handle Textbox.TextChanged in Grid
RadioButton3	Handle Textbox.TextChanged in Window

4. In the XAML for the *TextBox*, just before the />, type **TextChanged** and then press the Tab key twice. An entry for an event handler is created and a corresponding method is created in the code. The event-handler entry should look like the following:

```
TextChanged="TextBox1_TextChanged"
```

5. In the XAML for the *Grid*, type **TextBoxBase.TextChanged** and then press the Tab key twice to generate an event handler. The added XAML should look like this:

```
TextBoxBase.TextChanged="Grid_TextChanged"
```

6. In the XAML for the *Window*, type **TextBoxBase.TextChanged** and then press the Tab key twice to generate an event handler. The added XAML should look like this:

```
TextBoxBase.TextChanged="Window_TextChanged"
```

7. In Code view, add the following code to the *Textbox1_TextChanged* method:

```
' VB
MessageBox.Show("Event raised by Textbox")
e.Handled = RadioButton1.IsChecked

// C#
MessageBox.Show("Event raised by Textbox");
e.Handled = (bool)radioButton1.IsChecked;
```

8. Add the following code to the *Grid_TextChanged* method:

```
' VB
MessageBox.Show("Event raised by Grid")
e.Handled = RadioButton2.IsChecked

// C#
MessageBox.Show("Event raised by Grid");
e.Handled = (bool)radioButton2.IsChecked;
```

9. Add the following code to the *Window_TextChanged* method:

```
' VB
MessageBox.Show("Event raised by Window")
e.Handled = RadioButton3.IsChecked

// C#
MessageBox.Show("Event raised by Window");
e.Handled = (bool)radioButton3.IsChecked;
```

10. Press F5 to build and run your application. Type a letter in the *TextBox*. Three message boxes are displayed, each one indicating the control that raised the event. You can handle the event by choosing one of the radio buttons to halt event bubbling in the event handlers.

Lesson Summary

- WPF applications introduce a new kind of event called routed events. Routed events are raised by WPF controls.
- There are three kinds of routed events: direct, bubbling, and tunneling. Direct events are raised only by the control in which they originate. Bubbling and

tunneling events are raised by the control in which they originate and all controls that are higher in the visual tree.

- A tunneling event is raised first by the top-level control in the visual tree and tunnels down through the tree until it is finally raised by the control in which it originates. A bubbling event is raised first by the control in which the event originates and then bubbles up through the visual tree until it is finally raised by the top-level control in the visual tree.
- You can attach events that exist in contained controls to controls that are higher in the visual tree.
- The *EventManager* class exposes methods that allow you to manage events in your application. You can register a new routed event by using the *EventManager.RegisterRoutedEvent* class. You can create a class-level event handler by using *EventManager.RegisterClassHandler*.
- The *Application* object raises several events that can be handled to execute code at various points in the application's lifetime. You can handle application-level events in the code for the *Application* object.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Configuring Events and Event Handling." The questions are also available on the companion CD of this book if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. Suppose you have the following XAML code:

```
<Window x:Class="WpfApplication1.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300"
   ButtonBase.Click="Window_Click">
   <Grid ButtonBase.Click="Grid_Click">
      <StackPanel Margin="47,54,31,108" Name="stackPanel1"
         ButtonBase.Click="stackPanel1_Click">
         <Button Height="23" Name="button1" Width="75">Button</Button>
      </StackPanel>
   </Grid>
</Window>
```

Which method will be executed first when *button1* is clicked?

A. *Button1_Click*

B. *stackPanel1_Click*

C. *Grid_Click*

D. *Window_Click*

2. Suppose you have the following XAML code:

```
<Window x:Class="WpfApplication1.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300" MouseDown="Window_MouseDown">
   <Grid PreviewMouseDown="Grid_PreviewMouseDown">
      <StackPanel Margin="47,54,31,108" Name="stackPanel1"
         PreviewMouseDown="stackPanel1_PreviewMouseDown">
         <Button Click="button1_Click" Height="23" Name="button1"
            Width="75">Button</Button>
      </StackPanel>
   </Grid>
</Window>
```

Which method will be executed first when *button1* is clicked?

A. *Window_MouseDown*

B. *Grid_PreviewMouseDown*

C. *stackPanel1_PreviewMouseDown*

D. *button1_Click*

3. You are writing an application that consists of a single WPF window. You have code that you want to execute when the window first appears and every time the window is activated. What application event or events should you handle to accomplish this goal?

A. *Activated*

B. *Startup*

C. *Activated* and *Startup*

D. *Deactivated* and *Startup*

Lesson 2: Configuring Commands

WPF introduces new objects called *commands*. Commands represent high-level tasks that are performed in the application. For example, *Paste* is an example of a command—it represents the task of copying an object from the clipboard into a container. WPF provides a cohesive architecture for creating commands, associating them with application tasks, and hooking those commands up to user interface (UI) elements. In this lesson, you will learn to use the built-in command library, associate these commands with UI elements, define command handlers, add a gesture to a command, and define custom commands.

After this lesson, you will be able to:

- Explain the different parts of a command
- Associate a command with a UI element
- Add a gesture to a command
- Execute a command
- Associate a command with a command handler
- Disable a command
- Create a custom command

Estimated lesson time: 30 minutes

Commands, such as Cut, Copy, and Paste, represent tasks. In past versions of the .NET Framework, there was no complete architecture for associating code with tasks. For example, suppose you wanted to implement a *Paste* task in your application. You would create the code to execute the task, and then associate your UI element with that code via events. For example, you might have a *MenuItem* element that triggers the code when selected. You also might have context menu items and perhaps even a *Button* control. In past versions of the .NET Framework, you would have had to create event handlers for each control with which you want to associate the task. In addition, you would have had to implement code to inactivate each of these controls if the task was unavailable. While not an impossible task, doing this requires tedious coding that can be fraught with errors.

Commands allow you to use a centralized architecture for tasks. You can associate any number of UI controls or input gestures to a command and bind that command to a handler that is executed when controls are activated or gestures are performed. Commands also keep track of whether or not they are available. If a command is disabled, UI elements associated with that command are disabled, too.

Command architecture consists of four principal parts. There is the *Command* object itself, which represents the task. Then there are command sources. A *command source* is a control or gesture that triggers the command when invoked. The *command handler* is a method that is executed when the command is invoked, and *CommandBinding* is an object that is used by the .NET Framework to track what commands are associated with which sources and handlers.

The .NET Framework provides several predefined commands that are available for use by developers. These built-in commands are static objects that are properties of five static classes, which are the following:

- *ApplicationCommands*
- *ComponentCommands*
- *EditingCommands*
- *MediaCommands*
- *NavigationCommands*

Each of these classes exposes a variety of static command objects that you can use in your applications. While some of these commands have default input bindings (for example, the *ApplicationCommands.Open* command has a default binding to the key combination Ctrl+O), none of these commands has any inherent functionality—you must create bindings and handlers for these commands to use them in your application.

A High-Level Procedure for Implementing a Command

The following section describes a high-level procedure for implementing command functionality. The steps of this procedure are discussed in greater detail in the subsequent sections.

▶ **To implement a command**

1. Decide on the command to use, whether it is one of the static commands exposed by the .NET Framework or a custom command.
2. Associate the command with any controls in the user interface and add any desired input gestures to the command.
3. Create a method to handle the command.
4. Create a *CommandBinding* that binds the *Command* object to the command handler and optionally to a method that handles *Command.CanExecute*.

5. Add the command binding to the *Commands* collection of the control or *Window* where the command is invoked.

Invoking Commands

Once a command has been implemented, you can invoke it by associating it with a control, using a gesture, or invoking it directly from code.

Associating Commands with Controls

Many WPF controls implement the *ICommandSource* interface, which allows them to have a command associated with them that is fired automatically when that control is invoked. For example, *Button* and *MenuItem* controls implement *ICommandSource* and thus expose a *Command* property. When this property is set to a command, that command is executed automatically when the control is clicked. You can set a command for a control in XAML, as shown here:

```
<Button Command="ApplicationCommands.Find" Height="23"
HorizontalAlignment="Right" Margin="0,0,38,80" Name="Button3"
VerticalAlignment="Bottom" Width="75">Button</Button>
```

Invoking Commands with Gestures

You also can register mouse and keyboard gestures with *Command* objects that invoke the command when those gestures occur. The following example code shows how to add a mouse gesture and a keyboard gesture to the *InputGestures* collection of the *Application.Find* command:

```
' VB
ApplicationCommands.Find.InputGestures.Add(New _
   MouseGesture(MouseAction.LeftClick, ModifierKeys.Control))
ApplicationCommands.Find.InputGestures.Add(New _
   KeyGesture(Key.Q, ModifierKeys.Control))

// C#
ApplicationCommands.Find.InputGestures.Add(new
   MouseGesture(MouseAction.LeftClick, ModifierKeys.Control));
ApplicationCommands.Find.InputGestures.Add(new
   KeyGesture(Key.Q, ModifierKeys.Control));
```

Once the code in the previous example is executed, the *Find* command executes either when the Ctrl key is held down and the left mouse button is clicked, or when the Ctrl key and the Q key are held down together (Ctrl+Q).

Invoking Commands from Code

You might want to invoke a command directly from code, such as in response to an event in a control that does not expose a *Command* property. To invoke a command directly, simply call the *Command.Execute* method, as shown here:

```
' VB
ApplicationCommands.Find.Execute(aParameter, TargetControl)

// C#
ApplicationCommands.Find.Execute(aParameter, TargetControl);
```

In this example, *aParameter* represents an object that contains any required parameter data for the command. If no parameter is needed, you can use *null* (*Nothing* in Visual Basic). *TargetControl* is a control where the command originates. The run time will start looking for *CommandBindings* in this control and then bubble up through the visual tree until an appropriate *CommandBinding* is found.

Command Handlers and Command Bindings

As stated before, just invoking a command doesn't actually do anything. Commands represent tasks, but they do not contain any of the code for the tasks they represent. To execute code when a command is invoked, you must create a *CommandBinding* that binds the command to a command handler.

Command Handlers

Any method with the correct signature can be a command handler. Command handlers have the following signature:

```
' VB
Private Sub myCommandHandler(ByVal sender As Object, _
   ByVal e As ExecutedRoutedEventArgs)
   ' Handle the command here
End Sub

// C#
private void myCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
   // Handle the command here
}
```

ExecutedRoutedEventArgs is derived from *RoutedEventArgs* and thus exposes all the members that *RoutedEventArgs* does. In addition, it exposes a *Command* property that returns the *Command* object that is being handled.

Command Bindings

The *CommandBinding* object provides the glue that holds the whole command architecture together. A *CommandBinding* associates a command with a command handler. Adding a *CommandBinding* to the *CommandBindings* collection of the *Window* or a control registers the *CommandBinding* and allows the command handler to be called when the command is invoked. The following code demonstrates how to create and register a *CommandBinding:*

```
' VB
Dim abinding As New CommandBinding()
abinding.Command = ApplicationCommands.Find
AddHandler abinding.Executed, AddressOf myCommandHandler
Me.CommandBindings.Add(abinding)

// C#
CommandBinding abinding = new CommandBinding();
abinding.Command = ApplicationCommands.Find;
abinding.Executed += new ExecutedRoutedEventHandler(myCommandHandler);
this.CommandBindings.Add(abinding);
```

In the preceding example, you first create a new *CommandBinding* object. You then associate that *CommandBinding* object with a *Command* object. Next, you specify the command handler that will be executed when the command is invoked, and finally, you add the *CommandBinding* object to the *CommandBindings* collection of the *Window.* Thus, if an object in the window invokes the command, the corresponding command handler will be executed.

You also can define *CommandBindings* directly in the XAML. You can create a new binding and declaratively set the command it is associated with and the associated handlers. The following example demonstrates a new *CommandBinding* in the *CommandBinding* collection of the window that associates the *Application.Find* command with a handler:

```
<Window.CommandBindings>
   <CommandBinding Command="ApplicationCommands.Find"
      Executed="myCommandHandler" />
</Window.CommandBindings>
```

Command Bubbling

Note that all controls have their own *CommandBindings* collection in addition to the window's *CommandBindings* collection. This is because commands, like routed events, bubble up through the visual tree when they are invoked. Commands look for a binding first in the *CommandBindings* collection of the control in which they originate, and

then in the *CommandBindings* collections of controls higher on the visual tree. Like a *routedEvent,* you can stop further processing of the command by setting the *Handled* property of the *ExecutedRoutedEventArgs* parameter to *True*, as shown here:

```
' VB
Private Sub myCommandHandler(ByVal sender As Object, _
   ByVal e As ExecutedRoutedEventArgs)
   ' Stops further Command bubbling
   e.Handled = True
End Sub

// C#
private void myCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
   // Handle the command here
   e.Handled = true;
}
```

Exam Tip Bubbling and tunneling are concepts that are new to WPF and that play important roles both in commands and how WPF handles routed events. Be certain that you understand the concepts of bubbling and tunneling events and bubbling commands for the exam. Remember that a command or event doesn't need to be handled by the same element in which it originates.

Disabling Commands

Any command that is not associated with a *CommandBinding* is automatically disabled. No action is taken when that command is invoked, and any control that has its *Command* property set to that command appears as disabled. However, there might be times that you want to disable a command that is in place and associated with controls and *CommandBindings.* For example, you might want the Print command to be disabled until the focus is on a document. The command architecture allows you to designate a method to handle the *Command.CanExecute* event. The *CanExecute* event is raised at various points in the course of application execution to determine whether a command is in a state that will allow execution.

Methods that handle the *CanExecute* event include an instance of *CanExecuteRoutedEventArgs* as a parameter. This class exposes a property called *CanExecute* that is a boolean value. If *CanExecute* is true, the command can be invoked. If it is false, the command is disabled. You can create a method that handles the *CanExecute* event, determines whether or not the application is in an appropriate state to allow command execution, and sets *e.CanExecute* to the appropriate value.

▶ To handle the *CanExecute* event

1. Create a method to handle the *CanExecute* event. This method should query the application to determine whether the application's state is appropriate to allow the command to be enabled. An example is shown here:

```
' VB
Private canExecute As Boolean
Private Sub abinding_CanExecute(ByVal sender As Object, _
   ByVal e As CanExecuteRoutedEventArgs)
   e.CanExecute = canExecute
End Sub

// C#
bool canExecute;
void abinding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
   e.CanExecute = canExecute;
}
```

   In this example, the method returns the value represented by a private variable called *canExecute*. Presumably, the application sets this to False whenever it requires the command to be disabled.

2. Set the *CanExecute* handler on the *CommandBinding* to point to this method, as shown here:

```
' VB
' Assumes that you have already created a CommandBinding called abinding
AddHandler abinding.CanExecute, AddressOf abinding_CanExecute

// C#
// Assumes that you have already created a CommandBinding called abinding
abinding.CanExecute += new CanExecuteRoutedEventHandler(abinding_CanExecute);
```

   Alternatively, create a new binding in XAML and specify the handler there, as shown here in bold:

```
<Window.CommandBindings>
   <CommandBinding Command="ApplicationCommands.Find"
      Executed="CommandBinding_Executed"
      CanExecute="abinding_CanExecute" />
</Window.CommandBindings>
```

Creating Custom Commands

Although a wide variety of pre-existing commands is at your disposal, you might want to create your own custom commands. The best practice for custom commands is to follow the example set in the .NET Framework and create static classes (in C#) or

modules (in Visual Basic) that expose static instances of the custom command. This keeps multiple instances of the command from being created. You also can provide any custom configuration for the command in the static constructor of the class—for example, if you want to map any input gestures to the command. The following example shows how to create a static class that exposes a custom command called *Launch:*

```
' VB
Public Module MyCommands
   Private launch_command As RoutedUICommand
   Sub New()
      Dim myInputGestures As New InputGestureCollection
      myInputGestures.Add(New KeyGesture(Key.L, ModifierKeys.Control))
      launch_command = New RoutedUICommand("Launch", "Launch", _
        GetType(MyCommands), myInputGestures)
   End Sub
   Public ReadOnly Property Launch() As RoutedUICommand
      Get
         Return launch_command
      End Get
   End Property
End Module

// C#
public class MyCommands
{
   private static RoutedUICommand launch_command;
   static MyCommands()
   {
      InputGestureCollection myInputGestures = new
         InputGestureCollection();
      myInputGestures.Add(new KeyGesture(Key.L, ModifierKeys.Control));
      launch_command = new RoutedUICommand("Launch", "Launch",
         typeof(MyCommands), myInputGestures);
   }
   public RoutedUICommand Launch
   {
      get
      {
         return launch_command;
      }
   }
}
```

In this example, a static class or module is created to contain the custom command, which is exposed through a read-only property. In the static constructor, a new *InputGesturesCollection* is created and a key gesture is added to the collection. This collection is then used to initialize the instance of *RoutedUICommand* that is returned through the read-only property.

Using Custom Commands in XAML

Once you have created a custom command, you are ready to use it in code. If you want to use it in XAML, however, you also must map the namespace that contains the custom command to a XAML namespace. The following procedure describes how to use a custom command in XAML.

▶ **To use a custom command in XAML**

1. Create your custom command, as described previously.
2. Add a namespace mapping to your *Window* XAML. The following example demonstrates how to map a namespace called *WpfApplication13.CustomCommands*. Note that in this example, that would mean that your custom commands are kept in a separate namespace:

```
<Window x:Class="Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:CustomCommands="clr-namespace:WpfApplication13.CustomCommands"
   Title="Window1" Height="300" Width="300">


   <!-The rest of the XAML is omitted-->
</Window>
```

3. Use the newly mapped XAML namespace in your XAML code, as shown here:

```
<Button Command="CustomCommands:MyCommands.Launch" Height="23"
HorizontalAlignment="Left" Margin="60,91,0,0" Name="Button1" VerticalAlignment="Top"
Width="75">Button</Button>
```

Lab: Creating a Custom Command

In this lab, you create a custom command and then connect your command to UI elements by using a *CommandBinding*.

Exercise 1: Creating a Custom Command

1. From the CD, open the partial solution for this exercise.
2. From the Project menu, choose Add Class (in C#) or Add Module (in Visual Basic). Name the new item *CustomCommands* and click OK. Set the access modifier of this class or module to *public*.
3. If you are working in C#, add the following *using* statement to your class:

```
using System.Windows.Input;
```

   Otherwise, go on to Step 4.

4. Add a read-only property named *Launch* and a corresponding member variable that returns an instance of a *RoutedUICommand,* as shown here. (Note that these should be static members in C#.)

```
' VB
Private launch_command As RoutedUICommand
Public ReadOnly Property Launch() As RoutedUICommand
   Get
      Return launch_command
   End Get
End Property

// C#
private static RoutedUICommand launch_command;
public static RoutedUICommand Launch
{
   get
   {
      return launch_command;
   }
}
```

5. Add a constructor to your module (in Visual Basic) or a static constructor to your class (in C#) that creates a new *InputGestureCollection,* adds an appropriate input gesture to be associated with this new command, and then initializes the member variable that returns the custom command, as shown here:

```
' VB
Sub New()
   Dim myInputGestures As New InputGestureCollection
   myInputGestures.Add(New KeyGesture(Key.L, ModifierKeys.Control))
   launch_command = New RoutedUICommand("Launch", "Launch", _
      GetType(CustomCommands), myInputGestures)
End Sub

// C#
static CustomCommands()
{
   InputGestureCollection myInputGestures = new
      InputGestureCollection();
   myInputGestures.Add(new KeyGesture(Key.L, ModifierKeys.Control));
   launch_command = new RoutedUICommand("Launch", "Launch",
      typeof(CustomCommands), myInputGestures);
}
```

6. From the Build menu, choose Build Solution to build your solution.

Exercise 2: Using Your Custom Command

1. In XAML view, add the following code to your *Window* markup to create a reference to the class that contains your custom command:

   ```
   xmlns:Local="clr-namespace:YourProjectNamespaceGoesHere"
   ```

   The previous code in bold should be replaced with the namespace name of your project.

2. In XAML view, add the following attribute to both your *Button* control and your *Launch* MenuItem:

   ```
   Command="Local:CustomCommands.Launch"
   ```

3. In the *Window1* code view, add the following method:

   ```
   ' VB
   Private Sub Launch_Handler(ByVal sender As Object, _
      ByVal e As ExecutedRoutedEventArgs)
      MessageBox.Show("Launch invoked")
   End Sub

   // C#
   private void Launch_Handler(object sender, ExecutedRoutedEventArgs e)
   {
      MessageBox.Show("Launch invoked");
   }
   ```

4. From the Toolbox, drag a *CheckBox* control onto the form. Set the content of the control to "Enable Launch Command".

5. In the code view for *Window1*, add the following method:

   ```
   ' VB
   Private Sub LaunchEnabled_Handler(ByVal sender As Object, _
      ByVal e As CanExecuteRoutedEventArgs)
      e.CanExecute = CheckBox1.IsChecked
   End Sub

   // C#
   private void LaunchEnabled_Handler(object sender,
      CanExecuteRoutedEventArgs e)
   {
      e.CanExecute = (bool)checkBox1.IsChecked;
   }
   ```

6. Create or replace the constructor for *Window1* that creates and registers a *CommandBinding* for the *Launch* command. This *CommandBinding* should bind the *Launch.Executed* event to the *Launch_Handler* method and bind the *Launch.CanExecute* event to the *LaunchEnabled_Handler* method. An example is shown here:

```
' VB
Public Sub New()
   InitializeComponent()
   Dim abinding As New CommandBinding()
   abinding.Command = CustomCommands.Launch
   AddHandler abinding.Executed, AddressOf Launch_Handler
   AddHandler abinding.CanExecute, AddressOf LaunchEnabled_Handler
   Me.CommandBindings.Add(abinding)
End Sub

// C#
public Window1()
{
   InitializeComponent();
   CommandBinding abinding = new CommandBinding();
   abinding.Command = CustomCommands.Launch;
   abinding.Executed += new ExecutedRoutedEventHandler(Launch_Handler);
   abinding.CanExecute += new
      CanExecuteRoutedEventHandler(LaunchEnabled_Handler);
   this.CommandBindings.Add(abinding);
}
```

7. Press F5 to build and run your application. Note that when the application starts, the *Button* and *Launch* menu item are disabled. Select the check box to enable the command. Now you can invoke the command from the button, from the menu, or by using the Ctrl+L input gesture.

Lesson Summary

- Commands provide a central architecture for managing high-level tasks. The .NET Framework provides a library of built-in commands that map to common tasks that can be used in your applications.
- Commands can be invoked directly, by an input gesture such as a *MouseGesture* or a *KeyGesture*, or by activating a custom control. A single command can be associated with any number of gestures or controls.
- *CommandBindings* associate commands with command handlers. You can specify a method to handle the *Executed* event of a command and another method to handle the *CanExecute* event of a command.
- Methods handling the *CanExecute* event of a command should set the *CanExecute* property of the *CanExecuteRoutedEventArgs* to False when the command should be disabled.
- Commands can be bound by any number of *CommandBindings*. Commands exhibit bubbling behavior. When invoked, commands first look for a binding in

the collection of the element that the command was invoked in, and then look in each higher element in the visual tree.

- You can create custom commands. When you have created a custom command, you must map the namespace in which it exists to a XAML namespace in your XAML view.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Configuring Commands." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. Which of the following is required to bind a command to a command handler? (Choose all that apply.)
   - **A.** Instantiate a new instance of *CommandBinding*
   - **B.** Set the *CommandBinding.Command* property to a command
   - **C.** Add one or more input gestures to your command
   - **D.** Add a handler for the *CommandBinding.Executed* event
   - **E.** Add a handler for the *CommandBinding.CanExecute* event
   - **F.** Add *CommandBinding* to the *CommandBindings* collection of the *Window* or other control associated with the command
2. You are working with an application that exposes a command named *Launch*. This command is registered in the *CommandBindings* collection of a control called *Window11* and requires a *String* parameter. Which of the following code snippets invokes the command from code correctly?
   - **A.**

```
' VB
Launch.CanExecute = True
Launch.Execute("Boom", Window11)

// C#
Launch.CanExecute = true;
Launch.Execute("Boom", Window11);
```

B.

```
' VB
Launch.Execute("Boom")

// C#
Launch.Execute("Boom");
```

C.

```
' VB
Launch.Execute("Boom", Window11)

// C#
Launch.Execute("Boom", Window11);
```

D.

```
' VB
Window11.CanExecute(Launch, True)
Launch.Execute("Boom", Window11)

// C#
Window11.CanExecute(Launch, true);
Launch.Execute("Boom", Window11);
```

Lesson 3: Configuring Application Settings

The .NET Framework allows you to create and access values that persist from application session to application session. The values are called *settings*. Settings can represent any kind of information that an application might need from session to session, such as user preferences, the address of a Web server, or any other kind of necessary information. In this lesson, you will learn to create and access settings. You will learn the difference between a user setting and an application setting, and you will learn to load and save settings at run time.

After this lesson, you will be able to:

- Explain the difference between a user setting and an application setting
- Create a new setting at design time
- Load settings at run time
- Save user settings at run time

Estimated lesson time: 15 minutes

Settings can be used to store information that is valuable to the application but might change from time to time. For example, you can use settings to store user preferences, such as the color scheme of an application, or the address of a Web server used by the application.

Settings have four properties:

- *Name,* which indicates the name of the setting. This is used to access the setting at run time.
- *Type,* which represents the data type of the setting.
- *Value,* which is the value returned by the setting.
- *Scope,* which can be either *User* or *Application*.

The *Name, Type,* and *Value* properties should be fairly self-explanatory. The *Scope* property, however, bears a little closer examination. The *Scope* property can be set to either *Application* or *User*. A setting with *Application* scope represents a value that is used by the entire application regardless of the user, whereas an application with *User* scope is more likely to be user-specific and less crucial to the application.

An important distinction between user settings and application settings is that user settings are read/write. They can be read and written to at run time, and newly written

values can be saved by the application. In contrast, *Application settings* are read-only and the values can be changed only at design time or by editing the Settings file between application sessions.

Creating Settings at Design Time

Visual Studio provides an editor to create settings for your application at design time. This editor is shown in Figure 2-1.

Figure 2-1 The Settings Editor

The Settings Editor allows you to create new settings and set each of their four properties. The *Name* property, the name that you use to retrieve the setting value, must be unique in the application. The *Type* property represents the type of the setting. The *Scope* property is either *Application*, which represents a read-only property, or *User*, which represents a read-write setting. Finally, the *Value* property represents the value returned by the setting. The *Value* property must be of the type specified by the Type property.

▶ To create a setting at design time

1. If you are working in C#, in Solution Explorer, under Properties, locate and double-click Settings.settings to open the Settings Editor. If you are working in Visual Basic, in Solution Explorer, double-click MyProject and select the Settings tab.
2. Set the Name, Type, Scope, and Value for the new setting.
3. If your application has not yet been saved, choose Save All from the File menu to save your application.

Loading Settings at Run Time

At run time, you can access the values contained by the settings. In Visual Basic, settings are exposed through the *My* object, whereas in C#, you access settings through the *Properties.Settings.Default* object. At design time, individual settings appear in IntelliSense as properties of the *Settings* object and can be treated in code as such. Settings are strongly typed and are retrieved as the same type as specified when they were created. The following example code demonstrates how to copy the value from a setting to a variable:

```
' VB
Dim aString As String
aString = My.Settings.MyStringSetting

// C#
String aString;
aString = Properties.Settings.Default.MyStringSetting;
```

Saving User Settings at Run Time

You can save the value of user settings at run time. To change the value of a user setting, simply assign it a new value, just as you would any property or field. Then you must call the *Save* method to save the new value. An example is shown here:

```
' VB
My.Settings.Headline = "This is tomorrow's headline"
My.Settings.Save

// C#
Properties.Settings.Default.Headline = "This is tomorrow's headline";
Properties.Settings.Default.Save();
```

Quick Check

- What is the difference between a user setting and an application setting?

Quick Check Answer

- A user setting is designed to be user-specific, such as a background color. User settings can be written at run time and can vary from user to user. An application setting is designed to be constant for all users of an application, such as a database connection string. Application settings are read-only at run time.

Lab: Practice with Settings

In this lab you create an application that uses settings. You define settings while building the application, read the settings, apply them in your application, and enable the user to change one of the settings.

Exercise: Using Settings

1. In Visual Studio, create a new WPF application.
2. In Solution Explorer, expand Properties and double-click Settings.settings (in C#) or double-click My Project and choose the Settings tab (in Visual Basic) to open the Settings Editor.
3. Add two settings, as described in this table:

Name	Type	Scope	Value
ApplicationName	*String*	Application	Settings App
BackgroundColor	*System.Windows.Media.Color*	User	#ff0000ff

   Note that you will have to browse to find the *System.Windows.Media* type, then expand the node to find the *System.Windows.Media.Color* type.

4. In XAML view, add the following XAML to the *Grid* element to add a *ListBox* with four items and a *Button* to your user interface:

```
<ListBox Margin="15,15,0,0" Name="listBox1" Height="78"
   HorizontalAlignment="Left" VerticalAlignment="Top" Width="107">
   <ListBoxItem>Red</ListBoxItem>
   <ListBoxItem>Blue</ListBoxItem>
   <ListBoxItem>Green</ListBoxItem>
   <ListBoxItem>Tomato</ListBoxItem>
</ListBox>
<Button Margin="15,106,110,130" Name="button1">Change Background
   Color</Button>
```

5. In the designer, double-click *button1* to open the code view to the default handler for the *Click* event. Add the following code:

```
' VB
If Not listBox1.SelectedItem Is Nothing Then
   Dim astring As String = CType(listBox1.SelectedItem, _
      ListBoxItem).Content.ToString
   Select Case astring
      Case "Red"
         My.Settings.BackgroundColor = Colors.Red
```

```
      Case "Blue"
         My.Settings.BackgroundColor = Colors.Blue
      Case "Green"
         My.Settings.BackgroundColor = Colors.Green
      Case "Tomato"
         My.Settings.BackgroundColor = Colors.Tomato
   End Select
   Me.Background = New _
      System.Windows.Media.SolidColorBrush(My.Settings.BackgroundColor)
   My.Settings.Save()
End If

// C#
if (!(listBox1.SelectedItem == null))
{
   String astring =
      ((ListBoxItem)listBox1.SelectedItem).Content.ToString();
   switch (astring)
   {
      case "Red":
         Properties.Settings.Default.BackgroundColor = Colors.Red;
         break;
      case "Blue":
         Properties.Settings.Default.BackgroundColor = Colors.Blue;
         break;
      case "Green":
         Properties.Settings.Default.BackgroundColor = Colors.Green;
         break;
      case "Tomato":
         Properties.Settings.Default.BackgroundColor = Colors.Tomato;
         break;
   }
   this.Background = new
      System.Windows.Media.SolidColorBrush(
         Properties.Settings.Default.BackgroundColor);
   Properties.Settings.Default.Save();
}
```

6. Create or replace the constructor for this class with the following code to read and apply the settings:

```
' VB
Public Sub New()
   InitializeComponent()
   Me.Title = My.Settings.ApplicationName
   Me.Background = New _
      System.Windows.Media.SolidColorBrush(My.Settings.BackgroundColor)
End Sub

// C#
public Window1()
{
   InitializeComponent();
```

```
    this.Title = Properties.Settings.Default.ApplicationName;
    this.Background = new
        System.Windows.Media.SolidColorBrush(
            Properties.Settings.Default.BackgroundColor);
}
```

7. Press F5 to build and run your application. Note that the title of the window is the value of your *ApplicationName* setting and the background color of your window is the value indicated by the *BackgroundColor* setting. You can change the background color by selecting an item in the *ListBox* and clicking the button. After changing the background color, close the application and restart it. Note that the background color of the application at startup is the same as it was when the previous application session ended.

Lesson Summary

- Settings allow you to persist values between application sessions. You can add new settings at design time by using the Settings Editor.
- Settings can be one of two different scopes. Settings with Application scope are read-only at run time and can be changed only by altering the Settings file between application sessions. Settings with User scope are read-write at run time.
- You can access settings in code through *My.Settings* in Visual Basic, or *Properties.Settings.Default* in C#.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 3, "Configuring Application Settings." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. Which of the following code snippets correctly sets the value of a setting called *Title* and persists it?

   A.

```
' VB
My.Settings("Title") = "New Title"
My.Settings.Save
```

```
// C#
Properties.Settings.Default["Title"] = "New Title";
Properties.Settings.Default.Save();
```

B.

```
' VB
My.Settings("Title") = "New Title"

// C#
Properties.Settings.Default["Title"] = "New Title";
```

C.

```
' VB
My.Settings.Title = "New Title"
My.Settings.Save()

// C#
Properties.Settings.Default.Title = "New Title";
Properties.Settings.Default.Save();
```

D.

```
' VB
My.Settings.Title = "New Title"

// C#
Properties.Settings.Default.Title = "New Title";
```

2. Which of the following code snippets reads a setting of type *System.Windows.Media.Color* named *MyColor* correctly?

A.

```
' VB
Dim aColor As System.Windows.Media.Color
aColor = CType(My.Settings.MyColor, System.Windows.Media.Color)

// C#
System.Windows.Media.Color aColor;
aColor = (System.Windows.Media.Color)Properties.Settings.Default.MyColor;
```

B.

```
' VB
Dim aColor As System.Windows.Media.Color
aColor = My.Settings.MyColor.ToColor()

// C#
System.Windows.Media.Color aColor;
aColor = Properties.Settings.Default.MyColor.ToColor();
```

C.

```
' VB
Dim aColor As Object
aColor = My.Settings.MyColor

// C#
Object aColor;
aColor = Properties.Settings.Default.MyColor;
```

D.

```
' VB
Dim aColor As System.Windows.Media.Color
aColor = My.Settings.MyColor

// C#
System.Windows.Media.Color aColor;
aColor = Properties.Settings.Default.MyColor;
```

Chapter Review

To practice and reinforce the skills you learned in this chapter further, you can do any or all of the following:

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenarios. These scenarios set up real-world situations involving the topics of this chapter and ask you to create a solution.
- Complete the suggested practices.
- Take a practice test.

Chapter Summary

- Routed events can be raised by multiple UI elements in the visual tree. Bubbling events are raised first by the element in which they originate and then bubble up through the visual tree. Tunneling events are raised first by the topmost element in the visual tree and tunnel down to the element in which the event originates. Direct events are raised only by the element in which they originate.
- Elements in the visual tree can handle events that they do not themselves define. These are called *attached events*. You can define a handler for an attached event in the XAML that defines the element.
- You can use the *EventManager* class to register a new routed event and to register a class event handler.
- Commands provide an architecture that allows you to define high-level tasks, connect those tasks to a variety of inputs, define handlers that execute code when commands are invoked, and determine when a command is unavailable.
- You can use the built-in library of commands or create custom commands for your application. Commands can be triggered by controls, input gestures, or direct invocation.
- The *CommandBinding* object binds commands to command handlers.
- Settings allow you to create applications that persist between application sessions. Application scope settings are read-only at run time, and user scope settings are read-write at run time.
- Settings are exposed as strongly typed properties on the *My.Settings* object (in Visual Basic) and the *Properties.Settings.Default* object (in C#).

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- Application Setting
- Bubbling Event
- Command
- Command Handler
- Direct Event
- Event Handler
- Gesture
- Routed Event
- Setting
- Tunneling Event
- User Setting

Case Scenarios

In the following case scenarios, you will apply what you've learned about how to use commands, events, and settings to design user interfaces. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario 1: Validating User Input

You're creating a form that will be used by Humongous Insurance data entry personnel to input data. The form consists of several *TextBox* controls that receive input. Data entry is expected to proceed quickly and without errors, but to help ensure this you will be designing validation for this form. This validation is somewhat complex—there is a set of characters that is not allowed in any text box on the form, and each text box has additional limitations that differ from control to control. You would like to implement this validation scheme with a minimum of code in order to make troubleshooting and maintenance simple.

Question

Answer the following question for your manager:

- What strategies can we use to implement these requirements?

Case Scenario 2: Humongous Insurance User Interface

The front end for this database is just as complex as the validation requirements. You are faced with a front end that exposes many menu options. Furthermore, for expert users, some of the more commonly used menu items can be triggered by holding down the Ctrl key while performing various gestures with the mouse. Functionality invoked by the menu items sometimes will be unavailable. Finally, you need to allow the operator to edit data in this window quickly and easily.

Technical Requirements

- All main menu items must have access keys, and some have mouse shortcuts.
- Availability of menu items must be communicated to the user in a way that is easy to understand but does not disrupt program flow.
- You must ensure that when a menu item is unavailable, corresponding shortcut keys and mouse gestures are also inactivated.
- Certain *TextBox* controls on the form must fill in automatically when appropriate keystrokes are entered.

Question

- How can this functionality be implemented?

Suggested Practices

- Create a rudimentary text editor with buttons that implement the Cut, Copy, and Paste commands.
- Create an application that stores a color scheme for each user and automatically loads the correct color scheme when the user opens the application.
- Build an application that consists of a window with a single button that the user can chase around the window with the mouse but can never actually click.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the content covered in this chapter, or you can test yourself on all the 70-502 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO Practice tests

For details about all the practice test options available, see the section "How to Use the Practice Tests," in this book's Introduction.

Chapter 7

Styles and Animation

One of the major advances with the advent of the Windows Presentation Foundation (WPF) programming model is the uniquely agile use of the system's visual capabilities. In this chapter, you learn to use two aspects of WPF programming that take full advantage of these capabilities: styles and animation. Styles allow you to quickly apply changes to the visual interface and change the look and feel of your application in response to different conditions. Animations allow you to change property values over timelines that can be useful for a variety of visual effects. Together, these features allow you to harness the full power of the WPF presentation layer.

Exam objectives in this chapter:

- Create a consistent user interface appearance by using styles.
- Change the appearance of a UI element by using triggers.
- Add interactivity by using animations.

Lessons in this chapter:

- Lesson 1: Styles 305
- Lesson 2: Animations 323

Before You Begin

To complete the lessons in this chapter, you must have

- A computer that meets or exceeds the minimum hardware requirements listed in the "About This Book" section at the beginning of the book
- Microsoft Visual Studio 2008 Professional Edition installed on your computer
- An understanding of Microsoft Visual Basic or C# syntax and familiarity with Microsoft .NET Framework version 3.5
- An understanding of Extensible Application Markup Language (XAML)

Real World

Matthew Stoecker

At every turn, it seems that WPF provides more and more support for the creation of rich visual interfaces. The support for styles and triggers enables the rapid creation of interactive visual interfaces that used to take hours of coding event handlers. Likewise, animations now open up the possibilities for stunning user interfaces with minimal effort. I'm glad that now I can create attractive applications with the same amount of effort that the boxy old Windows Forms apps took!

Lesson 1: Styles

Styles allow you to create a cohesive look and feel for your application. You can use styles to define a standard color and sizing scheme for your application and use triggers to provide dynamic interaction with your UI elements. In this lesson, you learn to create and implement styles. You learn to apply a style to all instances of a single type and to implement style inheritance. You learn to use setters to set properties and event handlers, and you learn to use triggers to change property values dynamically. Finally, you learn about the order of property precedence.

After this lesson, you will be able to:

- Create and implement a style
- Apply a style to all instances of a type
- Implement style inheritance
- Use property and event setters
- Explain the order of property value precedence
- Use and implement triggers, including property triggers, data triggers, event triggers, and multiple triggers

Estimated lesson time: 30 minutes

Using Styles

Styles can be thought of as analogous to cascading style sheets as used in Hypertext Markup Language (HTML) pages. Styles basically tell the presentation layer to substitute a new visual appearance for the standard one. They allow you to make changes to the user interface as a whole easily and to provide a consistent look and feel for your application in a variety of situations. Styles enable you to set properties and hook up events on UI elements through the application of those styles. Further, you can create visual elements that respond dynamically to property changes through the application of triggers, which listen for a property change and then apply style changes in response.

Properties of Styles

The primary class in the application of styles is, unsurprisingly, the *Style* class. The *Style* class contains information about styling a group of properties. A *Style* can be created to

apply to a single instance of an element, to all instances of an element type, or across multiple types. The important properties of the *Style* class are shown in Table 7-1.

Table 7-1 Important Properties of the *Style* Class

Property	Description
BasedOn	Indicates another style that this style is based on. This property is useful for creating inherited styles.
Resources	Contains a collection of local resources used by the style. The *Resources* property is discussed in detail in Chapter 9, "Resources, Documents, and Localization."
Setters	Contains a collection of *Setter* or *EventSetter* objects. These are used to set properties or events on an element as part of a style.
TargetType	This property identifies the intended element type for the style.
Triggers	Contains a collection of *Trigger* objects and related objects that allow you to designate a change in the user interface in response to changes in properties.

The basic skeleton of a <*Style*> element in XAML markup looks like the following:

```
<Style>
   <!-- A collection of setters is enumerated here -->
   <Style.Triggers>
   <!-- A collection of Trigger and related objects is enumerated here -->
   </Style.Triggers>
   <Style.Resources>
      <!-- A collection of local resources for use in the style -->
   </Style.Resources>
</Style>
```

Setters

The most common class you will use in the construction of Styles is the *Setter*. As their name implies, *Setters* are responsible for setting some aspect of an element. *Setters* come in two flavors: property setters (or just *Setters*, as they are called in markup), which set values for properties; and event setters, which set handlers for events.

Property Setters

Property setters, represented by the <*Setter*> tag in XAML, allow you to set properties of elements to specific values. A property setter has two important properties: the *Property* property, which designates the property that is to be set by the *Setter,* and the *Value* property, which indicates the value to which the property is to be set. The following example demonstrates a *Setter* that sets the Background property of a *Button* element to *Red:*

```
<Setter Property="Button.Background" Value="Red" />
```

The value for the *Property* property must take the form of the following:

```
Element.PropertyName
```

If you want to create a style that sets a property on multiple different types of elements, you could set the style on a common class that the elements inherit, as shown here:

```
<Style>
   <Setter Property="Control.Background" Value="Red" />
</Style>
```

This style sets the *Background* property of all elements that inherit from the *Control* to which it is applied.

Event Setters

Event setters (represented by the <*EventSetter*> tag) are similar to property setters, but they set event handlers rather than property values. The two important properties for an *EventSetter* are the *Event* property, which specifies the event for which the handler is being set; and the *Handler* property, which specifies the event handler to attach to that event. An example is shown here:

```
<EventSetter Event="Button.MouseEnter" Handler="Button_MouseEnter" />
```

The value of the *Handler* property must specify an extant event handler with the correct signature for the type of event with which it is connected. Similar to property setters, the format for the *Event* property is

```
Element.EventName
```

where the element type is specified, followed by the event name.

Creating a Style

You've seen the simplest possible implementation of a style: a single *Setter* between two *Style* tags, but you haven't yet seen how to apply a style to an element. There are several ways to apply a style to an element or elements. This section examines the various ways to apply a style to elements in your user interface.

Setting the *Style* Property Directly

The most straightforward way to apply a style to an element is to set the *Style* property directly in XAML. The following example demonstrates directly setting the *Style* property of a *Button* element:

```
<Button Height="25" Name="Button1" Width="100">
   <Button.Style>
      <Style>
         <Setter Property="Button.Content" Value="Style set directly" />
         <Setter Property="Button.Background" Value="Red" />
      </Style>
   </Button.Style>
</Button>
```

While setting the *Style* directly in an element might be the most straightforward, it is seldom the best method. When setting the *Style* directly, you must set it for each element that you want to be affected by the *Style*. In most cases, it is simpler to set the properties of the element directly at design time.

One scenario where you might want to set the *Style* directly on an element is to provide a set of *Triggers* for that element. Because *Triggers* must be set in a *Style* (except for *EventTriggers*, as you will see in the next section), you conceivably could set the *Style* directly to set triggers for an element.

Setting a Style in a *Resources* Collection

The most common method for setting styles is to create the style as a member of a *Resources* collection and then apply the style to elements in your user interface by referencing the resource. The following example demonstrates creating a style as part of the *Windows.Resources* collection:

```
<Window.Resources>
   <Style x:Key="StyleOne">
      <Setter Property="Button.Content" Value="Style defined in resources" />
      <Setter Property="Button.Background" Value="Red" />
   </Style>
</Window.Resources>
```

Under most circumstances, you must supply a key value for a *Style* that you define in the *Resources* collection. Then you can apply that style to an element by referencing the resource, as shown in bold here:

```
<Button Name="Button1" Style="{StaticResource StyleOne}" Height="30"
   Width="200" />
```

The advantage to defining a *Style* in the *Resources* section is that you can then apply that *Style* to multiple elements by simply referencing the resource. Resources are discussed in detail in Chapter 9.

Applying Styles to All Controls of a Specific Type

You can use the *TargetType* property to specify a type of element to be associated with the style. When you set the *TargetType* property on a *Style,* that *Style* is applied to all elements of that type automatically. Further, you do not need to specify the qualifying type name in the *Property* property of any *Setters* that you use—you can just refer to the property name. When you specify the *TargetType* for a *Style* that you have defined in a *Resources* collection, you do not need to provide a key value for that style. The following example demonstrates the use of the *TargetType* property:

```
<Window.Resources>
   <Style TargetType="Button">
      <Setter Property=" Content" Value="Style set for all buttons" />
      <Setter Property="Background" Value="Red" />
   </Style>
</Window.Resources>
```

When you apply the *TargetType* property, you do not need to add any additional markup to the elements of that type to apply the style.

If you want an individual element to opt out of the style, you can set the style on that element explicitly, as seen here:

```
<Button Style="{x:Null}" Margin="10">No Style</Button>
```

This example explicitly sets the *Style* to *Null,* which causes the *Button* to revert to its default look. You also can set the *Style* to another *Style* directly, as seen earlier in this lesson.

Setting a Style Programmatically

You can create and define a style programmatically. While defining styles in XAML is usually the best choice, creating a style programmatically might be useful when you want to create and apply a new style dynamically, possibly based on user preferences.

The typical method for creating a style programmatically is to create the *Style* object in code; then create *Setters* (and *Triggers*, if appropriate); add them to the appropriate collection on the *Style* object; and then when finished, set the *Style* property on the target elements. The following example demonstrates creating and applying a simple style in code:

```
' VB
Dim aStyle As New Style
Dim aSetter As New Setter
aSetter.Property = Button.BackgroundProperty
aSetter.Value = Brushes.Red
aStyle.Setters.Add(aSetter)
Dim bSetter As New Setter
bSetter.Property = Button.ContentProperty
bSetter.Value = "Style set programmatically"
aStyle.Setters.Add(bSetter)
Button1.Style = aStyle

// C#
Style aStyle = new Style();
Setter aSetter = new Setter();
aSetter.Property = Button.BackgroundProperty;
aSetter.Value = Brushes.Red;
aStyle.Setters.Add(aSetter);
Setter bSetter = new Setter();
bSetter.Property = Button.ContentProperty;
bSetter.Value = "Style set programmatically";
aStyle.Setters.Add(bSetter);
Button1.Style = aStyle;
```

You can also define a style in a *Resources* collection and apply that style in code, as shown here:

```
<!-- XAML -->
<Window.Resources>
   <Style x:Key="StyleOne">
      <Setter Property="Button.Content" Value="Style applied in code" />
      <Setter Property="Button.Background" Value="Red" />
   </Style>
</Window.Resources>

' VB
Dim aStyle As Style
aStyle = CType(Me.Resources("StyleOne"), Style)
Button1.Style = aStyle

// C#
Style aStyle;
aStyle = (Style)this.Resources["StyleOne"];
Button1.Style = aStyle;
```

Implementing Style Inheritance

You can use inheritance to create styles that conform to the basic look and feel of the original style but provide differences that offset some controls from others. For example, you might create one *Style* for all the *Button* elements in your user interface and create an inherited style to provide emphasis for one of the buttons. You can use the *BasedOn* property to create *Style* objects that inherit from other *Style* objects. The *BasedOn* property references another style and automatically inherits all the members of that *Style* and then allows you to build on that *Style* by adding additional members. The following example demonstrates two *Style* objects—an original *Style* and a *Style* that inherits it:

```
<Window.Resources>
   <Style x:Key="StyleOne">
      <Setter Property="Button.Content" Value="Style set in original Style" />
      <Setter Property="Button.Background" Value="Red" />
      <Setter Property="Button.FontSize" Value="15" />
      <Setter Property="Button.FontFamily" Value="Arial" />
   </Style>
   <Style x:Key="StyleTwo" BasedOn="{StaticResource StyleOne}">
      <Setter Property="Button.Content" Value="Style set by inherited style" />
      <Setter Property="Button.Background" Value="AliceBlue" />
      <Setter Property="Button.FontStyle" Value="Italic" />
   </Style>
</Window.Resources>
```

The result of applying these two styles is seen in Figure 7-1.

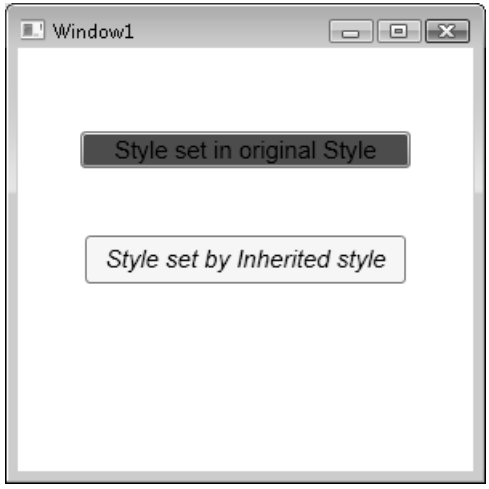


Figure 7-1 Two buttons—the original and an inherited style

When a property is set in both the original style and the inherited style, the property value set by the inherited style always takes precedence. But when a property is set by the original style and not set by the inherited style, the original property setting is retained.

Quick Check

- Under what circumstances is a *Style* automatically applied to an element? How else can a *Style* be applied to an element?

Quick Check Answer

- A *Style* is applied to an element automatically when it is declared as a resource in the page and the *TargetType* property of the *Style* is set. If the *TargetType* property is not set, you can apply a *Style* to an element by setting that element's *Style* property, either in XAML or in code.

Triggers

Along with *Setters*, *Triggers* make up the bulk of objects that you use in creating styles. *Triggers* allow you to implement property changes declaratively in response to other property changes that would have required event-handling code in Windows Forms programming. There are five kinds of *Trigger* objects, as listed in Table 7-2.

Table 7-2 Types of *Trigger* Objects

Type	Class Name	Description
Property trigger	*Trigger*	Monitors a property and activates when the value of that property matches the *Value* property.
Multi-trigger	*MultiTrigger*	Monitors multiple properties and activates only when all the monitored property values match their corresponding *Value* properties.
Data trigger	*DataTrigger*	Monitors a bound property and activates when the value of the bound property matches the *Value* property.

Table 7-2 Types of *Trigger* Objects

Type	Class Name	Description
Multi-data-trigger	*MultDataTrigger*	Monitors multiple bound properties and activates only when all the monitored bound properties match their corresponding *Value* properties.
Event trigger	*EventTrigger*	Initiates a series of *Actions* when a specified event is raised.

A *Trigger* is active only when it is part of a *Style.Triggers* collection—with one exception. *EventTrigger* objects can be created within a *Control.Triggers* collection outside a *Style*. The *Control.Triggers* collection can accommodate only *EventTriggers*, and any other *Trigger* placed in this collection causes an error. *EventTriggers* are primarily used with animation and are discussed further in Lesson 2 of this chapter, "Animations."

Property Triggers

The most commonly used type of *Trigger* is the property trigger. The property trigger monitors the value of a property specified by the *Property* property. When the value of the specified property equals the *Value* property, the *Trigger* is activated. Important properties of property triggers are shown in Table 7-3.

Table 7-3 Important Properties of Property Triggers

Property	Description
EnterActions	Contains a collection of *Action* objects that are applied when the *Trigger* becomes active. Actions are discussed in greater detail in Lesson 2 of this chapter.
ExitActions	Contains a collection of *Action* objects that are applied when the *Trigger* becomes inactive. *Actions* are discussed in greater detail in Lesson 2 of this chapter.
Property	Indicates the property that is monitored for changes.
Setters	Contains a collection of *Setter* objects that are applied when the *Trigger* becomes active.
Value	Indicates the value that is compared to the property referenced by the *Property* property.

Triggers listen to the property indicated by the *Property* property and compare that property to the *Value* property. When the referenced property and the *Value* property are equal, the *Trigger* is activated. Any *Setter* objects in the *Setters* collection of the *Trigger* are applied to the style, and any *Actions* in the *EnterActions* collections are initiated. When the referenced property no longer matches the *Value* property, the *Trigger* is inactivated. All *Setter* objects in the *Setters* collection of the *Trigger* are inactivated, and any *Actions* in the *ExitActions* collection are initiated.

NOTE *Actions* are used primarily in animations, and they are discussed in greater detail in Lesson 2 of this chapter.

The following example demonstrates a simple *Trigger* object that changes the *FontWeight* of a *Button* element to *Bold* when the mouse enters the *Button:*

```
<Style.Triggers>
   <Trigger Property="Button.IsMouseOver" Value="True">
      <Setter Property="Button.FontWeight" Value="Bold" />
   </Trigger>
</Style.Triggers>
```

In this example, the *Trigger* defines one *Setter* in its *Setters* collection. When the *Trigger* is activated, that *Setter* is applied.

Multi-triggers

Multi-triggers are similar to property triggers in that they monitor the value of properties and activate when those properties meet a specified value. The difference is that multi-triggers are capable of monitoring several properties at a single time and they activate only when all monitored properties equal their corresponding *Value* properties. The properties that are monitored and their corresponding *Value* properties are defined by a collection of *Condition* objects. The following example demonstrates a *MultiTrigger* that sets the *Button.FontWeight* property to *Bold* only when the *Button* is focused and the mouse has entered the control:

```
<Style.Triggers>
   <MultiTrigger>
      <MultiTrigger.Conditions>
         <Condition Property="Button.IsMouseOver" Value="True" />
         <Condition Property="Button.IsFocused" Value="True" />
      </MultiTrigger.Conditions>
      <MultiTrigger.Setters>
         <Setter Property="Button.FontWeight" Value="Bold" />
      </MultiTrigger.Setters>
   </MultiTrigger>
</Style.Triggers>
```

Data Triggers and Multi-data-triggers

Data triggers are similar to property triggers in that they monitor a property and activate when the property meets a specified value, but they differ in that the property they monitor is a bound property. Instead of a *Property* property, data triggers expose a *Binding* property that indicates the bound property to listen to. The following shows a data trigger that changes the *Background* property of a *Label* to *Red* when the bound property *CustomerName* equals "Fabrikam":

```
<Style.Triggers>
   <DataTrigger Binding="{Binding Path=CustomerName}" Value="Fabrikam">
      <Setter Property="Label.Background" Value="Red" />
   </DataTrigger>
</Style.Triggers>
```

Multi-data-triggers are to data triggers as multi-triggers are to property triggers. They contain a collection of *Condition* objects, each of which specifies a bound property via its *Binding* property and a value to compare to that bound property. When all the conditions are satisfied, the *MultiDataTrigger* activates. The following example demonstrates a *MultiDataTrigger* that sets the *Label.Background* property to *Red* when *CustomerName* equals "Fabrikam" and *OrderSize* equals 500:

```
<Style.Triggers>
   <MultiDataTrigger>
      <MultiDataTrigger.Conditions>
         <Condition Binding="{Binding Path=CustomerName}" Value="Fabrikam" />
         <Condition Binding="{Binding Path=OrderSize}" Value="500" />
      </MultiDataTrigger.Conditions>
      <MultiDataTrigger.Setters>
         <Setter Property="Label.Background" Value="Red" />
      </MultiDataTrigger.Setters>
   </MultiDataTrigger>
</Style.Triggers>
```

Event Triggers

Event triggers are different from the other *Trigger* types. While other *Trigger* types monitor the value of a property and compare it to an indicated value, event triggers specify an event and activate when that event is raised. In addition, event triggers do not have a *Setters* collection—rather, they have an *Actions* collection. Although you have been exposed briefly to the *SoundPlayerAction* in Chapter 4, "Adding and Managing Content," most actions deal with animations, which are discussed in detail in Lesson 2 of this chapter. The following two examples demonstrate the *EventTrigger*

class. The first example uses a *SoundPlayerAction* to play a sound when a *Button* is clicked:

```
<EventTrigger RoutedEvent="Button.Click">
   <SoundPlayerAction Source="C:\myFile.wav" />
</EventTrigger>
```

The second example demonstrates a simple animation that causes the *Button* to grow in height by 200 units when clicked:

```
<EventTrigger RoutedEvent="Button.Click">
   <EventTrigger.Actions>
      <BeginStoryboard>
         <Storyboard>
            <DoubleAnimation Duration="0:0:5"
               Storyboard.TargetProperty="Height" To="200" />
         </Storyboard>
      </BeginStoryboard>
   </EventTrigger.Actions>
</EventTrigger>
```

Understanding Property Value Precedence

By now, you have probably noticed that a property can be set in many different ways. They can be set in code; they can be set by styles; they can have default values; and so on. It might seem logical at first to believe that a property will have the value it was last set to, but this is actually incorrect. There is a defined and strict order of precedence that determines a property's value based on *how* it was set, not when. The precedence order is summarized here, with highest precedence listed first:

1. Set by coercion by the property system.
2. Set by active animations or held animations.
3. Set locally, either by code, by direct setting in XAML, or through data binding.
4. Set by the *TemplatedParent*. Within this category, there is a sub-order of precedence, again listed in descending order:
   a. Set by *Triggers* from the templated parent
   b. Set by the templated parent through property sets
5. Implicit style—this applies only to the *Style* property.
6. Set by *Style* triggers.
7. Set by *Template* triggers.

8. Set by *Style* setters.
9. Set by the default *Style*. There is a sub-order within this category, again listed in descending order:
    a. Set by *Triggers* in the default style
    b. Set by *Setters* in the default style
10. Set by inheritance.
11. Set by metadata.

Exam Tip The order of property precedence seems complicated, but actually it is fairly logical. Be sure that you understand the concept behind the property order in addition to knowing the order itself.

This may seem like a complicated and arbitrary order of precedence, but upon closer examination it is actually very logical and based upon the needs of the application and the user. The highest precedence is property coercion. This takes place in some elements if an attempt is made to set a property beyond its allowed values. For example, if an attempt is made to set the *Value* property of a *Slider* control to a value higher than the *Maximum* property, the *Value* is coerced to equal the *Maximum* property. Next in precedence come animations. For animations to have any meaningful use, they must be able to override preset property values. The next highest level of precedence is properties that have been set explicitly through developer or user action.

Properties set by the *TemplatedParent* are next in the order of precedence. These are properties set on objects that come into being through a template. Templates are discussed further in Chapter 8, "Customizing the User Interface." After this comes a special precedence item that applies only to the *Style* property of an element: Provided that the *Style* property has not been set by any item with a higher-level precedence, it is set to a *Style* whose *TargetType* property matches the type of the element in question. Then come properties set by *Triggers*—first those set by a *Style,* then those set by a *Template*. This is logical because for triggers to have any meaningful effect, they must override properties set by styles.

Properties set by styles come next: first properties set by user-defined styles, and then properties set by the default style (also called the *Theme,* which typically is set by the operating system). Finally come properties that are set through inheritance and the application of metadata.

For developers, there are a few important implications that are not intuitively obvious. The most important is that if you set a property explicitly—whether in XAML or in code—the explicitly set property blocks any changes dictated by a *Style* or *Trigger*. WPF assumes that you want that property value to be there for a reason and does not allow it to be set by a *Style* or *Trigger,* although it still can be overridden by an active animation.

A second, less obvious implication is that when using the Visual Studio designer to drag and drop items onto the design surface from the *ToolBox,* the designer explicitly sets several properties, especially layout properties. These property settings have the same precedence as they would if you had set them yourself. So if you are designing a style-oriented user interface, you should either enter XAML code directly in XAML view to create controls and set as few properties explicitly as possible, or you should review the XAML that Visual Studio generates and delete settings as appropriate.

You can clear a property value that has been set in XAML or code manually by calling the *DependencyObject.ClearValue* method. The following code example demonstrates how to clear the value of the *Width* property on a button named *Button1*:

```
' VB
Button1.ClearValue(WidthProperty)

// C#
Button1.ClearValue(WidthProperty);
```

Once the value has been cleared, it can be reset automatically by the property system.

Lab: Creating High-Contrast Styles

In this lab, you create a rudimentary high-contrast *Style* for *Button, TextBox,* and *Label* elements.

Exercise 1: Using Styles to Create High-Contrast Elements

1. Create a new WPF application in Visual Studio.
2. In XAML view, just above the <*Grid*> declaration, create a *Window.Resources* section, as shown here:

```
<Window.Resources>

</Window.Resources>
```

3. In the *Window.Resources* section, create a high-contrast *Style* for *TextBox* controls that sets the background color to *Black* and the foreground to *White*. The *TextBox* controls also should be slightly larger by default. An example is shown here:

```
<Style TargetType="TextBox">
   <Setter Property="Background" Value="Black" />
   <Setter Property="Foreground" Value="White" />
   <Setter Property="BorderBrush" Value="White" />
   <Setter Property="Width" Value="135" />
   <Setter Property="Height" Value="30" />
</Style>
```

4. Create similar styles for *Button* and *Label*, as shown here:

```
<Style TargetType="Label">
   <Setter Property="Background" Value="Black" />
   <Setter Property="Foreground" Value="White" />
   <Setter Property="Width" Value="135" />
   <Setter Property="Height" Value="33" />
</Style>
<Style TargetType="Button">
   <Setter Property="Background" Value="Black" />
   <Setter Property="Foreground" Value="White" />
   <Setter Property="Width" Value="135" />
   <Setter Property="Height" Value="30" />
</Style>
```

5. Type the following in XAML view. Note that you should not add controls from the toolbox because that automatically sets some properties in the designer at a higher property precedence than styles:

```
<Label Margin="26,62,126,0" VerticalAlignment="Top">
   High-Contrast Label</Label>
<TextBox Margin="26,117,126,115">High-Contrast TextBox
   </TextBox>
<Button Margin="26,0,126,62" VerticalAlignment="Bottom">
   High-Contrast Button</Button>
```

6. Press F5 to build and run your application. Note that while the behavior of these controls is unaltered, their appearance has changed.

Exercise 2: Using Triggers to Enhance Visibility

1. In XAML view for the solution you completed in Exercise 1, add a *Style.Triggers* section to the *TextBox Style*, as shown here:

```
<Style.Triggers>

</Style.Triggers>
```

2. In the *Style.Triggers* section, add *Triggers* that detect when the mouse is over the control and enlarge the *FontSize* of the control, as shown here:

```
<Trigger Property="IsMouseOver" Value="True">
   <Setter Property="FontSize" Value="20" />
</Trigger>
```

3. Add similar *Style.Triggers* collections to your other two styles.
4. Press F5 to build and run your application. The *FontSize* of a control now increases when you move the mouse over it.

Lesson Summary

- *Styles* allow you to define consistent visual styles for your application. *Styles* use a collection of *Setters* to apply style changes. The most commonly used *Setter* type is the property setter, which allows you to set a property. Event setters allow you to hook up event handlers as part of an applied style.
- *Styles* can be set inline, but more frequently, they are defined in a *Resources* collection and are set by referring to the resource. You can apply a style to all instances of a control by setting the *TargetType* property to the appropriate type.
- *Styles* are most commonly applied declaratively, but they can be applied in code by creating a new style dynamically or obtaining a reference to a preexisting *Style* resource.
- You can create styles that inherit from other styles by using the *BasedOn* property.
- Property triggers monitor the value of a dependency property and can apply *Setters* from their *Setters* collection when the monitored property equals a predetermined value. Multi-triggers monitor multiple properties and apply their *Setters* when all monitored properties match corresponding specified values. Data triggers and multi-data-triggers are analogous but monitor bound values instead of dependency properties.
- Event triggers perform a set of *Actions* when a particular event is raised. They are used most commonly to control *Animations*.
- Property values follow a strict order of precedence depending on how they are set.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Styles." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

1. Look at the following XAML snippet:

```
<Window.Resources>
   <Style x:Key="Style1">
      <Setter Property="Label.Background" Value="Blue" />
      <Setter Property="Button.Foreground" Value="Red" />
      <Setter Property="Button.Background" Value="LimeGreen" />
   </Style>
</Window.Resources>
<Grid>
   <Button Height="23" Margin="81,0,122,58" Name="Button1"
      VerticalAlignment="Bottom">Button</Button>
</Grid>
```

   Assuming that the developer hasn't set any properties any other way, what is the *Background* color of *Button1?*

   A. Blue

   B. Red

   C. LimeGreen

   D. System Default

2. Look at the following XAML snippet:

```
<Window.Resources>
   <Style x:Key="Style1">
      <Style.Triggers>
         <MultiTrigger>
            <MultiTrigger.Conditions>
               <Condition Property="TextBox.IsMouseOver"
                  Value="True" />
               <Condition Property="TextBox.IsFocused"
                  Value="True" />
            </MultiTrigger.Conditions>
            <Setter Property="TextBox.Background"
               Value="Red" />
         </MultiTrigger>
      </Style.Triggers>
   </Style>
</Window.Resources>
<Grid>
   <TextBox Style="{StaticResource Style1}" Height="21"
      Margin="75,0,83,108" Name="TextBox1"
      VerticalAlignment="Bottom" />
</Grid>
```

When will *TextBox1* appear with a red background?

A. When the mouse is over *TextBox1*

B. When *TextBox1* is focused

C. When *TextBox1* is focused and the mouse is over *TextBox1*

D. All of the above

E. Never

3. Look at the following XAML snippet:

```
<Window.Resources>
   <Style TargetType="Button">
      <Setter Property="Content" Value="Hello" />
      <Style.Triggers>
         <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Content" Value="World" />
         </Trigger>
         <Trigger Property="IsMouseOver" Value="False">
            <Setter Property="Content" Value="How are you?" />
         </Trigger>
      </Style.Triggers>
   </Style>
</Window.Resources>
<Grid>
   <Button Height="23" Margin="81,0,122,58" Name="Button1"
      VerticalAlignment="Bottom">Button</Button>
</Grid>
```

What does *Button1* display when the mouse is NOT over the *Button*?

A. *Hello*

B. *World*

C. *Button*

D. *How are you?*

Lesson 2: Animations

Animations are another new feature of WPF. Animations allow you to change the value of a property over the course of a set period of time. Using this technique, you can create a variety of visual effects, including causing controls to grow or move abut the user interface, to change color gradually, or to change other properties over time. In this lesson, you learn how to create animations that animate a variety of property types and use *Storyboard* objects to control the playback of those animations.

After this lesson, you will be able to:

- Create and use animations
- Control animations with the *Storyboard* class
- Control timelines and playback of animations
- Implement simultaneous animations
- Use *Actions* to control animation playback
- Implement animations that use key frames
- Create and start animations in code

Estimated lesson time: 30 minutes

Using Animations

The term *animation* brings to mind hand-drawn anthropomorphic animals performing amusing antics in video media, but in WPF, animation has a far simpler meaning. Generally speaking, an animation in WPF refers to an automated property change over a set period of time. You can animate an element's size, location, color, or virtually any other property or properties associated with an element. You can use the *Animation* classes to implement these changes.

The *Animation* classes are a large group of classes designed to implement these automated property changes. There are 42 *Animation* classes in the *System.Windows.Media.Animation* namespace, and each one has a specific data type that they are designed to animate. *Animation* classes fall into three basic groups: Linear animations, key frame–based animations, and path-based animations.

Linear animations, which automate a property change in a linear way, are named in the format *<TypeName>Animation*, where *<TypeName>* is the name of the type being animated. *DoubleAnimation* is an example of a linear animation class, and that is the animation class you are likely to use the most.

Key frame–based animations perform their animation on the basis of several waypoints, called key frames. The flow of a key-frame animation starts at the beginning, and then progresses to each of the key frames before ending. The progression is usually linear. Key-frame animations are named in the format *<TypeName>AnimationUsingKeyFrames*, where *<TypeName>* is the name of the *Type* being animated. An example is *StringAnimationUsingKeyFrames*.

Path-based animations use a *Path* object to guide the animation. They are used most often to animate properties that relate to the movement of visual objects along a complex course. Path-based animations are named in the format *<TypeName>AnimationUsingPath*, where *<TypeName>* is the name of the type being animated. There are currently only three path-based *Animation* classes—*PointAnimationUsingPath*, *DoubleAnimationUsingPath*, and *MatrixAnimationUsingPath*.

Important Properties of Animations

Although there are many different *Animation* classes, they all work in the same fundamental way—they change the value of a designated property over a period of time. As such, they share common properties. Many of these properties also are shared with the *Storyboard* class, which is used to organize *Animation* objects, as you will see later in this lesson. Important common properties of the *Animation* and *Storyboard* classes are shown in Table 7-4.

Table 7-4 Important Properties of the *Animation* and *Storyboard* Classes

Property	Description
AccelerationRatio	Gets or sets a value specifying the percentage of the *Duration* property of the *Animation* that is spent accelerating the passage of time from zero to its maximum rate.
AutoReverse	Gets or sets a value that indicates whether the *Animation* plays in reverse after it completes a forward iteration.
BeginTime	Gets or sets the time at which the *Animation* should begin, relative to the time that the *Animation* is executed. For example, an *Animation* with a *BeginTime* set to 0:0:5 exhibits a 5-second delay before beginning.
DecelerationRatio	Gets or sets a value specifying the percentage of the duration of the *Animation* spent decelerating the passage of time from its maximum rate to zero.

Table 7-4 Important Properties of the *Animation* and *Storyboard* Classes

Property	Description
Duration	Gets or sets the length of time for which the *Animation* plays.
FillBehavior	Gets or sets a value that indicates how the *Animation* behaves after it has completed.
RepeatBehavior	Gets or sets a value that indicates how the *Animation* repeats.
SpeedRatio	Gets or sets the rate at which the *Animation* progresses relative to its parent.

In addition, the linear animation classes typically implement a few more important properties, which are described in Table 7-5.

Table 7-5 Important Properties of Linear Animation Classes

Property	Description
From	Gets or sets the starting value of the *Animation*. If omitted, the *Animation* uses the current property value.
To	Gets or sets the ending value of the *Animation*.
By	Gets or sets the amount by which to increase the value of the target property over the course of the *Animation*. If both the *To* and *By* properties are set, the value of the *By* property is ignored.

The following example demonstrates a very simple animation. This animation changes the value of a property that has a *Double* data type representation from 1 to 200 over the course of 10 seconds:

```
<DoubleAnimation Duration="0:0:10" From="1" To="200" />
```

In this example, the *Duration* property specifies a duration of 10 seconds for the animation, and the *From* and *To* properties indicate a starting value of 1 and an ending value of 200.

You might notice that something seems to be missing from this example. What property is this animation animating? The answer is that it is not animating any property—the *Animation* object carries no intrinsic information about the property that is being animated, but instead it is applied to a property by means of a *Storyboard*.

Storyboard Objects

The *Storyboard* is the object that controls and organizes animations in your user interface. The *Storyboard* class contains a *Children* collection, which organizes a collection of *Timeline* objects, which include *Animation* objects. When created declaratively in XAML, all *Animation* objects must be enclosed within a *Storyboard* object, as shown here:

```
<Storyboard>
   <DoubleAnimation Duration="0:0:10" From="1" To="200" />
</Storyboard>
```

Using a *Storyboard* to Control Animations

In XAML, *Storyboard* objects organize your *Animation* objects. The most important feature of the *Storyboard* object is that it contains properties that allow you to specify the target element and target property of the child *Animation* objects, as shown in bold in this example:

```
<Storyboard TargetName="Button1" TargetProperty="Height">
   <DoubleAnimation Duration="0:0:10" From="1" To="200" />
</Storyboard>
```

This example is now usable. It defines a timeline where over the course of 10 seconds, the *Height* property of *Button1* goes from a value of 1 to a value of 200.

The *TargetName* and *TargetProperty* properties are attached properties, so instead of defining them in the *Storyboard* itself, you can define them in the child *Animation* objects, as shown in bold here:

```
<Storyboard>
   <DoubleAnimation Duration="0:0:10" From="1" To="200"
      Storyboard.TargetName="Button1"
      Storyboard.TargetProperty="Height" />
</Storyboard>
```

Because a *Storyboard* can hold more than one *Animation* at a time, this configuration allows you to set separate target elements and properties for each animation. Thus, it is more common to use the attached properties.

Simultaneous Animations

The *Storyboard* can contain multiple child *Animation* objects. When the *Storyboard* is activated, all child animations are started at the same time and run simultaneously.

The following example demonstrates two simultaneous *Animations* that cause both the *Height* and *Width* of a *Button* element to grow over 10 seconds:

```
<Storyboard>
   <DoubleAnimation Duration="0:0:10" From="1" To="200"
      Storyboard.TargetName="Button1"
      Storyboard.TargetProperty="Height" />
   <DoubleAnimation Duration="0:0:10" From="1" To="100"
      Storyboard.TargetName="Button1"
      Storyboard.TargetProperty="Widtht" />
</Storyboard>
```

Using *Animations* with *Triggers*

You now have learned most of the story about using *Animation* objects. The *Animation* object defines a property change over time, and the *Storyboard* object contains *Animation* objects and determines what element and property the *Animation* objects affect. But there is still one piece that is missing: How do you start and stop an *Animation?*

All declaratively created *Animation* objects must be housed within a *Trigger* object. This can be either as a part of a *Style,* or in the *Triggers* collection of an *Element,* which accepts only *EventTrigger* objects.

Trigger objects define collections of *Action* objects, which control when an *Animation* is started and stopped. The following example demonstrates an *EventTrigger* object with an inline *Animation:*

```
<EventTrigger RoutedEvent="Button.Click">
   <EventTrigger.Actions>
      <BeginStoryboard>
         <Storyboard>
            <DoubleAnimation Duration="0:0:5"
               Storyboard.TargetProperty="Height" To="200" />
         </Storyboard>
      </BeginStoryboard>
   </EventTrigger.Actions>
</EventTrigger>
```

As you can see in the preceding example, the *Storyboard* object is enclosed in a *BeginStoryboard* tag, which itself is enclosed in the *EventTrigger.Actions* tag. *BeginStoryboard* is an *Action*–it indicates that the contained *Storyboard* should be started. The *EventTrigger* class defines a collection of *Actions* that should be initiated when the *Trigger* is activated, and in this example, *BeginStoryboard* is the action that is initiated. Thus, when the *Button* indicated in this trigger is clicked, the described *Animation* runs.

Using *Actions* to Control Playback

There are several *Action* classes that can be used to manage animation playback. These classes are summarized in Table 7-6.

Table 7-6 Animation-Related *Action* Classes

Action	Description
BeginStoryboard	Begins the child *Storyboard* object.
PauseStoryboard	Pauses the playback of an indicated *Storyboard* at the current playback position.
ResumeStoryboard	Resumes playback of an indicated *Storyboard*.
SeekStoryboard	Fast-forwards to a specified position in a target *Storyboard*.
SetStoryboardSpeedRatio	Sets the *SpeedRatio* of the specified *Storyboard*.
SkipStoryboardToFill	Moves the specified *Storyboard* to the end of its timeline.
StopStoryboard	Stops playback of the specified *Storyboard* and returns the animation to the starting position.

PauseStoryboard, ResumeStoryboard, SkipStoryboardToFill, and *StopStoryboard* are all fairly self-explanatory. They cause the indicated *Storyboard* to pause, resume, stop, or skip to the end, as indicated by the *Action* name. The one property that all these *Action* classes have in common is the *BeginStoryboardName* property. This property indicates the name of the *BeginStoryboard* object that the action is to affect. The following example demonstrates a *StopStoryboard* action that stops the *BeginStoryBoard* object named *stb1:*

```
<Style.Triggers>
   <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
         <BeginStoryboard Name="stb1">
            <Storyboard>
               <DoubleAnimation Duration="0:0:5"
                  Storyboard.TargetProperty="Height" To="200" />
            </Storyboard>
         </BeginStoryboard>
      </EventTrigger.Actions>
   </EventTrigger>
   <EventTrigger RoutedEvent="Button.MouseLeave">
```

```
      <EventTrigger.Actions>
         <StopStoryboard BeginStoryboardName="stb1" />
      </EventTrigger.Actions>
   </EventTrigger>
</Style.Triggers>
```

All *Actions* that affect a particular *Storyboard* object must be defined in the same *Triggers* collection. The previous example shows both of these triggers being defined in the *Button.Triggers* collection. If you were to define these triggers in separate *Triggers* collections, storyboard actions would not function.

The *SetStoryboardSpeedRatio* action sets the speed ratio for the entire *Storyboard* and all *Animation* objects in that *Storyboard*. In addition to *BeginStoryboardName*, you must set the *SpeedRatio* property of this *Action* as well. The following example demonstrates a *SetStoryboardSpeedRatio* action that speeds the referenced *Storyboard* by a factor of 2:

```
<Style.Triggers>
   <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
         <BeginStoryboard Name="stb1">
            <Storyboard>
               <DoubleAnimation Duration="0:0:5"
                  Storyboard.TargetProperty="Height" To="200" />
            </Storyboard>
         </BeginStoryboard>
      </EventTrigger.Actions>
   </EventTrigger>
   <EventTrigger RoutedEvent="Button.MouseLeave">
      <EventTrigger.Actions>
         <SetStoryboardSpeedRatio BeginStoryboardName="stb1" SpeedRatio="2" />
      </EventTrigger.Actions>
   </EventTrigger>
</Style.Triggers>
```

The *SeekStoryboard* action requires two additional properties to be set. The *Origin* property can be either a value of *BeginTime* or of *Duration* and specifies how the *Offset* property is applied. An *Origin* value of *BeginTime* specifies that the *Offset* is relative to the beginning of the *Storyboard*. An *Origin* value of *Duration* specifies that the *Offset* is relative to the *Duration* property of the *Storyboard*. The *Offset* property determines the amount of the offset to jump to in the animation. The following example shows a *SeekStoryboard* action that skips the referenced timeline to 5 seconds ahead from its current point in the timeline.

```
<Style.Triggers>
   <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
```

```
            <BeginStoryboard Name="stb1">
               <Storyboard>
                  <DoubleAnimation Duration="0:0:10"
                     Storyboard.TargetProperty="Height" To="200" />
               </Storyboard>
            </BeginStoryboard>
         </EventTrigger.Actions>
      </EventTrigger>
      <EventTrigger RoutedEvent="Button.MouseLeave">
         <EventTrigger.Actions>
            <SeekStoryboard BeginStoryboardName="stb1" Origin="BeginTime"
               Offset="0:0:5" />
         </EventTrigger.Actions>
      </EventTrigger>
   </Style.Triggers>
```

Using Property Triggers with *Animations*

In the examples shown in this section, you have seen *Actions* being hosted primarily in *EventTrigger* objects. You can also host *Action* objects in other kinds of *Triggers. Trigger, MultiTrigger, DataTrigger, and MultiDataTrigger* objects host two *Action* collections: *EnterActions* and *ExitActions* collections.

The *EnterActions* collection hosts a set of *Actions* that are executed when the *Trigger* is activated. Conversely, the *ExitActions* collection hosts a set of *Actions* that are executed when the *Trigger* is deactivated. The following demonstrates a *Trigger* that begins a *Storyboard* when activated and stops that *Storyboard* when deactivated:

```
<Trigger Property="IsMouseOver" Value="True">
   <Trigger.EnterActions>
      <BeginStoryboard Name="stb1">
         <Storyboard>
            <DoubleAnimation Storyboard.TargetProperty="FontSize"
               To="20" Duration="0:0:.5" />
         </Storyboard>
      </BeginStoryboard>
   </Trigger.EnterActions>
   <Trigger.ExitActions>
      <StopStoryboard BeginStoryboardName="stb1" />
   </Trigger.ExitActions>
</Trigger>
```

Managing the Playback Timeline

Both the *Animation* class and the *Storyboard* class contain several properties that allow you to manage the playback timeline with a fine level of control. Each of these properties is discussed in this section. When a property is set on an *Animation,* the setting affects only that animation. Setting a property on a *Storyboard,* however, affects all *Animation* objects it contains.

AccelerationRatio and *DecelerationRatio*

The *AccelerationRatio* and *DecelerationRatio* properties allow you to designate a part of the timeline for acceleration and deceleration of the animation speed, rather than starting and playing at a constant speed. This is used sometimes to give an animation a more "natural" appearance. These properties are expressed in fractions of 1 and represent a percentage value of the total timeline. Thus, an *AccelerationRatio* with a value of .2 indicates that 20 percent of the timeline should be spent accelerating to the top speed. So the *AccelerationRatio* and *DecelerationRatio* properties should be equal to or less than 1 when added together. This example shows an *Animation* with an *AccelerationRatio* of .2:

```
<DoubleAnimation Duration="0:0:5" AccelerationRatio="0.2"
   Storyboard.TargetProperty="Height" To="200" />
```

AutoReverse

As the name implies, the *AutoReverse* property determines whether the animation automatically plays out in reverse after the end is reached. A value of *True* indicates that the *Animation* will play in reverse after the end is reached. *False* is the default value. The following example demonstrates this property:

```
<DoubleAnimation Duration="0:0:5" AutoReverse="True"
   Storyboard.TargetProperty="Height" To="200" />
```

FillBehavior

The *FillBehavior* property determines how the *Animation* behaves after it has completed. A value of *HoldEnd* indicates that the *Animation* holds the final value after it has completed, whereas a value of *Stop* indicates that the *Animation* stops and returns to the beginning of the timeline when completed. An example is shown here:

```
<DoubleAnimation Duration="0:0:5" FillBehavior="Stop"
   Storyboard.TargetProperty="Height" To="200" />
```

The default value for *FillBehavior* is *HoldEnd*.

RepeatBehavior

The *RepeatBehavior* property determines if and how an animation repeats. The *RepeatBehavior* property can be set in three ways. First, it can be set to *Forever,* which indicates that an *Animation* repeats for the duration of the application. Second, it can be set to a number followed by the letter *x* (for example, 2*x*), which indicates the number of times to repeat the animation. Third, it can be set to a *Duration,* which indicates the amount of time that an *Animation* plays, irrespective of the number of iterations. The following three examples demonstrate these settings. The first demonstrates an *Animation* that

repeats forever, the second an *Animation* that repeats three times, and the third an *Animation* that repeats for 1 minute:

```
<DoubleAnimation Duration="0:0:5" RepeatBehavior="Forever"
   Storyboard.TargetProperty="Height" To="200" />
<DoubleAnimation Duration="0:0:5" RepeatBehavior="3x"
   Storyboard.TargetProperty="Height" To="200" />
<DoubleAnimation Duration="0:0:5" RepeatBehavior="0:1:0"
   Storyboard.TargetProperty="Height" To="200" />
```

SpeedRatio

The *SpeedRatio* property allows you to speed up or slow down the base timeline. The *SpeedRatio* value represents the coefficient for the speed of the *Animation*. Thus, an *Animation* with a *SpeedRatio* value of 0.5 takes twice the standard time to complete, whereas a value of 2 causes the *Animation* to complete twice as fast. An example is shown here:

```
<DoubleAnimation Duration="0:0:5" SpeedRatio="0.5"
   Storyboard.TargetProperty="Height" To="200" />
```

Animating Non-Double Types

Most of the examples that you have seen in this lesson have dealt with the *DoubleAnimation* class, but in fact a class exists for every animatable data type. For example, the *ColorAnimation* class allows you to animate a color change, as shown here:

```
<Button Height="23" Width="100" Name="Button1">
   <Button.Background>
      <SolidColorBrush x:Name="myBrush" />
   </Button.Background>
   <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
         <BeginStoryboard>
            <Storyboard>
               <ColorAnimation Storyboard.TargetName="myBrush"
                  Storyboard.TargetProperty="Color" From="Red" To="LimeGreen"
                  Duration="0:0:5" />
            </Storyboard>
         </BeginStoryboard>
      </EventTrigger>
   </Button.Triggers>
</Button>
```

In this example, when the button is clicked, the background color of the button gradually changes from red to lime green over the course of 5 seconds.

NOTE In the standard Windows theme, this animation may conflict with other animations in the button's default template, so you might need to mouse out of the button and defocus it to see the full effect.

Animation with Key Frames

Up until now, all the animations you have seen have used linear interpolation—that is, the animated property changes take place over a linear timeline at a linear rate. You also can create nonlinear animations by using key frames.

Key frames are waypoints in an animation. Instead of allowing the *Animation* to progress linearly from beginning to end, key frames divide the animation up into short segments. The animation progresses from the beginning to the first key frame, then the next, and through the *KeyFrames* collection until the end of the animation is reached. Each key frame defines its own *Value* and *KeyTime* properties, which indicate the value that the *Animation* will represent when it reaches the key frame and the time in the *Animation* at which that frame will be reached.

Every data type that supports a linear *Animation* type also supports a key-frame *Animation* type, and some types that do not have linear animation types have key-frame *Animation* types. The key-frame *Animation* types are named *<TargetType>AnimationUsingKeyFrames*, where *<TargetType>* represents the name of the *Type* animated by the *Animation*. Key-frame *Animation* types do not support the *From*, *To*, and *By* properties; rather, the course of the *Animation* is defined by the collection of key frames.

There are three different kinds of key frames. The first is linear key frames, which are named *Linear<TargetType>KeyFrame*. These key frames provide points in an *Animation* that are interpolated between in a linear fashion. The following example demonstrates the use of linear key frames:

```
<DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Height">
   <LinearDoubleKeyFrame Value="10" KeyTime="0:0:1" />
   <LinearDoubleKeyFrame Value="100" KeyTime="0:0:2" />
   <LinearDoubleKeyFrame Value="30" KeyTime="0:0:4"/>
</DoubleAnimationUsingKeyFrames>
```

In the preceding example, the *Height* property goes from its starting value to a value of 10 in the first second, then to a value of 100 in the next second, and finally returns to a value of 30 in the last 2 seconds. The progression between each segment is interpolated linearly. In this example, it is similar to having several successive linear *Animation* objects.

Discrete Key Frames

Some animatable data types do not support gradual transitions under any circumstances. For example, the *String* type can only accept discrete changes. You can use discrete key frame objects to make discrete changes in the value of an animated property. Discrete key frame classes are named *Discrete<TargetType>KeyFrame*, where *<TargetType>* is the *Type* being animated. Like linear key frames, discrete key frames use a *Value* and a *KeyTime* property to set the parameters of the key frame. The following example demonstrates an animation of a *String* using discrete key frames:

```
<StringAnimationUsingKeyFrames Storyboard.TargetProperty="Content">
   <DiscreteStringKeyFrame Value="Soup" KeyTime="0:0:0" />
   <DiscreteStringKeyFrame Value="Sous" KeyTime="0:0:1" />
   <DiscreteStringKeyFrame Value="Sots" KeyTime="0:0:2" />
   <DiscreteStringKeyFrame Value="Nots" KeyTime="0:0:3" />
   <DiscreteStringKeyFrame Value="Nuts" KeyTime="0:0:4" />
</StringAnimationUsingKeyFrames>
```

Spline Key Frames

Spline key frames allow you to define a Bézier curve that expresses the relationship between animation speed and animation time, thus allowing you to create animations that accelerate and decelerate in complex ways. While the mathematics of Bézier curves is beyond the scope of this lesson, a Bézier curve is simply a curve between two points whose shape is influenced by two control points. Using spline key frames, the start and end points of the curve are always (0,0) and (1,1) respectively, so you must define the two control points. The *KeySpline* property accepts two points to define the Bézier curve, as seen here:

```
<SplineDoubleKeyFrame Value="300" KeyTime="0:0:6" KeySpline="0.1,0.8 0.6,0.6" />
```

Spline key frames are difficult to create with the intended effect without complex design tools, and are most commonly used when specialized animation design tools are available.

Using Multiple Types of Key Frames in an Animation

You can use multiple types of key frames in a single animation—you can freely intermix *LinearKeyFrame*, *DiscreteKeyFrame*, and *SplineKeyFrame* objects in the *KeyFrames* collection. The only restriction is that all key frames you use must be appropriate to the *Type* that is being animated. *String* animations, for example, can use only *DiscreteStringKeyFrame* objects.

Quick Check

- What are the different types of key frame objects? When would you use each one?

Quick Check Answer

- There are *LinearKeyFrame*, *DiscreteKeyFrame*, and *SplineKeyFrame* objects. *LinearKeyFrame* objects indicate a linear transition from the preceding property value to the value represented in the key frame. *DiscreteKeyFrame* objects represent a sudden transition from the preceding property value to the value represented in the key frame. *SplineKeyFrame* objects represent a transition whose rate is defined by the sum of an associated Bézier curve. You would use each of these types when the kind of transition represented was the kind of transition that you wanted to incorporate into your user interface. In addition, some animation types can use only *DiscreteKeyFrames*.

Creating and Starting Animations in Code

All the *Animation* objects that you have seen so far in this lesson were created declaratively in XAML. However, you can create and execute *Animation* objects just as easily in code as well.

The process of creating an *Animation* should seem familiar to you; as with other .NET objects, you create a new instance of your *Animation* and set the relevant properties, as seen in this example:

```
' VB
Dim aAnimation As New System.Windows.Media.Animation.DoubleAnimation()
aAnimation.From = 20
aAnimation.To = 300
aAnimation.Duration = New Duration(New TimeSpan(0, 0, 5))
aAnimation.FillBehavior = Animation.FillBehavior.Stop

// C#
System.Windows.Media.Animation.DoubleAnimation aAnimation = new
   System.Windows.Media.Animation.DoubleAnimation();
aAnimation.From = 20;
aAnimation.To = 300;
aAnimation.Duration = new Duration(new TimeSpan(0, 0, 5));
aAnimation.FillBehavior = Animation.FillBehavior.Stop;
```

After the *Animation* has been created, however, the obvious question is: How do you start it? When creating *Animation* objects declaratively, you must use a *Storyboard* to

organize your *Animation* and an *Action* to start it. In code, however, you can use a simple method call. All WPF controls expose a method called *BeginAnimation,* which allows you to specify a dependency property on that control and an *Animation* object to act on that dependency property. The following code shows an example:

```
' VB
Button1.BeginAnimation(Button.HeightProperty, aAnimation)

// C#
button1.BeginAnimation(Button.HeightProperty, aAnimation);
```

Lab: Improving Readability with Animations

In this lab, you improve upon your solution to the lab in Lesson 1 of this chapter. You remove the triggers that cause the *FontSize* to expand and instead use an *Animation* to make it look more natural. In addition, you create *Animation* objects to increase the size of the control when the mouse is over it.

Exercise: Animating High-Contrast Styles

1. Open the completed solution from the lab from Lesson 1 of this chapter.
2. In each of the *Styles,* remove the *FontSize Setter* that is defined in the *Trigger* and replace it with a *Trigger.EnterActions* and *Trigger.ExitActions* section, as shown here:

```
<Trigger.EnterActions>

</Trigger.EnterActions>
<Trigger.ExitActions>

</Trigger.ExitActions>
```

3. In each *Trigger.EnterActions* section, add a *BeginStoryboard* action, as shown here:

```
<BeginStoryboard Name="Storyboard1">

</BeginStoryboard>
```

4. Add the following *Storyboard* and *Animation* objects to the *BeginStoryboard* object in the style for the *TextBox.* Note that the values for the *ThicknessAnimation* object are crafted specifically for the completed version of the Lesson 1 lab on the CD. If you created your own solution, you need to recalculate these values:

```
<Storyboard Duration="0:0:1">
   <DoubleAnimation Storyboard.TargetProperty="FontSize"
      To="20" />
   <ThicknessAnimation Storyboard.TargetProperty="Margin"
      To="26,118,45,104" />
```

```
   <DoubleAnimation Storyboard.TargetProperty="Width" To="210"/>
   <DoubleAnimation Storyboard.TargetProperty="Height" To="40"/>
</Storyboard>
```

5. Add a similar *Storyboard* to the style for the *Label,* as shown here:

```
<Storyboard Duration="0:0:1">
   <DoubleAnimation Storyboard.TargetProperty="FontSize" To="20" />
   <ThicknessAnimation Storyboard.TargetProperty="Margin"
      To="26,62,46,-10" />
   <DoubleAnimation Storyboard.TargetProperty="Width" To="210"/>
   <DoubleAnimation Storyboard.TargetProperty="Height" To="40"/>
</Storyboard>
```

6. Add a similar *Storyboard* to the style for the *Button,* as shown here:

```
<Storyboard Duration="0:0:1">
   <DoubleAnimation Storyboard.TargetProperty="FontSize" To="20" />
   <ThicknessAnimation Storyboard.TargetProperty="Margin"
      To="26,0,46,52" />
   <DoubleAnimation Storyboard.TargetProperty="Width" To="210"/>
   <DoubleAnimation Storyboard.TargetProperty="Height" To="40"/>
</Storyboard>
```

7. Add the following line to the *Trigger.ExitActions* section of each *Style:*

```
<StopStoryboard BeginStoryboardName="Storyboard1" />
```

8. Press F5 to build and run your application. Now the *FontSize* expansion is animated and the control expands as well.

Lesson Summary

- Animation objects drive automated property changes over time. There are three different types of *Animation* objects—linear animations, key frame–based animations, and path-based animations. Every animatable type has at least one *Animation* type associated with it, and some types have more than one type of *Animation* that can be applied.
- Storyboard objects organize one or more *Animation* objects. Storyboard objects determine what objects and properties their contained *Animation* objects are applied to.
- Both *Animation* and *Storyboard* objects contain a variety of properties that control *Animation* playback behavior.
- Storyboard objects that are created declaratively are activated by a *BeginStoryboard* action in the *Actions* collection of a *Trigger. Triggers* also can define actions that pause, stop, and resume *Storyboard* objects, as well as performing other *Storyboard*-related functions.

- Key frame animations define a series of waypoints through which the *Animation* passes. There are three kinds of key frames: linear key frames, discrete key frames, and spline key frames. Some animatable types, such as *String*, support only discrete key frames.
- You can create and apply *Animation* objects in code. When doing this, you do not need to define a *Storyboard* object; rather, you call the *BeginAnimation* method on the element with which you want to associate the *Animation*.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Animations." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. How many times does the *Animation* shown here repeat (not counting the first iteration)?

```
<DoubleAnimation Duration="0:0:15" RepeatBehavior="0:1:0"
   Storyboard.TargetProperty="Height" To="200" />
```

   A. 0

   B. 1

   C. 2

   D. 3

2. Look at this *Animation:*

```
<DoubleAnimation Duration="0:0:5" From="30" By="80" To="200"
Storyboard.TargetProperty="Height" />
```

   Assuming that the element whose *Height* property it animates begins with a *Height* of 50, what is the value of the element after the animation has completed?

   A. 50

   B. 110

   C. 130

   D. 200

Chapter Review

To practice and reinforce the skills you learned in this chapter further, you can do any or all of the following:

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenarios. These scenarios set up real-world situations involving the topics of this chapter and ask you to create a solution.
- Complete the suggested practices.
- Take a practice test.

Chapter Summary

- *Styles* allow you to define consistent visual styles for your application by using a collection of *Setters*. They usually are defined as a *Resource* and referenced in XAML, though they can be set inline or dynamically. *Styles* can be inherited from other styles and applied to all instances of a particular type.
- Triggers respond to changes in the application environment. Property triggers and multi-triggers listen for changes in property values, and data triggers and multi-data-triggers listen for changes in bound values. When one of these triggers is activated, its *Setters* collection is applied. *EventTriggers* listen for a routed event and execute *Actions* in response to that event.
- Property values follow a strict order of precedence depending on how they are set.
- *Animation* objects drive automated property changes over time. There are three different types of *Animation* objects—linear animations, key frame–based animations, and path-based animations. Every animatable type has one or more *Animation* classes that can be used with it. *Animations* are organized by *Storyboard* objects, which are themselves controlled by *Action* objects that are activated in the *Action* collections of *Trigger* objects.
- *Animations* that use key frames provide waypoints that the *Animation* visits as it progresses. Key frames can be linear, spline-based, or discrete.
- You can create and apply *Animation* objects in code. When doing this, you do not need to define a *Storyboard* object, but rather you call the *BeginAnimation* method on the element with which you want to associate the *Animation*.

Key Terms

- Action
- Animation
- Key Frame
- Setter
- Storyboard
- Style
- Trigger

Case Scenarios

In the following case scenarios, you apply what you've learned about how to use controls to design user interfaces. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario 1: Cup Fever

You've had a little free time around the office, and you decided to write a simple but snazzy application to organize and display results from World Cup soccer matches. The technical details are all complete: You've located a Web service that feeds up-to-date scores, and you've created a database that automatically applies updates from this service for match results and keeps track of upcoming matches. The database is exposed through a custom data object built on *ObservableCollection*<> lists. All that remains are the finishing touches. Specifically, when users choose an upcoming match from a drop-down box at the top of the window, you want the window's color scheme to match the colors of the teams in the selected matchup.

Technical Requirements

- The user interface is divided into two sections, each of which is built on a Grid container. Each section represents a team in the current or upcoming match. The user interface for each section must apply the appropriate team colors automatically when a new match is chosen.

Question

Answer the following question for all your office mates, who are eagerly awaiting the application's completion.

- How can you implement these color changes to the user interface?

Case Scenario 2: A Far-Out User Interface

Our friends with the questionable taste are back. They were so impressed with the work you did for them back in Chapter 4 that they've asked you to design a user interface that further pushes the envelope of good design sensibilities. Rather than having a static tie-dyed appearance, now they want the background to be a constantly changing multicolored experience. The idea of using a *RadialGradientBrush* to paint the background of the window is still acceptable, but they want the center of the gradient to change over time and they want the colors of the background to change.

Question

Answer the following question for your manager:

- How can we implement this appearance?

Suggested Practices

- Create an *Animation* that moves elements across the user interface. Alternatively, use linear animations and key frame animations to explore a variety of different animation styles. Animate other properties of UI elements as well, such as the color, size, and content.
- Use *Animations* to create a slideshow application that reads all the image files in a given directory and displays each image for 10 seconds before automatically switching to the next one. Note that you have to create and apply the *Animation* in code.
- Modify the solution from Lesson 2 of Chapter 6, "Converting and Validating Data," to create styles for the application that includes *DataTriggers* that automatically apply styles based on the *CompanyName* of the selected record.
- Modify the solution from the second lab in this chapter to reverse the *Animation* instead of stopping it when the mouse exits the control.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the content covered in this chapter, or you can test yourself on all the 70-502 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO Practice tests

For details about all the practice test options available, see the section "How to Use the Practice Tests," in this book's Introduction.

Index

Symbols and Numbers

.csv file, 430
.msi files, 445
.NET Framework, 288
  application events, 67
  commands, 72–73
  event architecture, 59
  Navigation applications, 9
  tunneling events, 61
.NET properties, 373
.wav files, 176–79
.xaml files
  Application, 393
  Classic.xaml, 379
  Generic.xaml, 376, 379

A

AccelerationRatio, 324, 331
access control
  file system, 11–12, 14
  objects, 48–49
  registry, 11, 14
  XBAPs, 11, 14
access keys, 104
Action, 315–16, 327–29, 335–36
Activated events, 67
AddBackEntry, 26
AddExtension, 346
AddHandler, 28
ADO.NET, 213, 226–29
Aero.NormalColor.xaml, 379
aligning content, 144–45
AllowsTransparency, 5, 7
alphanumeric characters, 352
Alt key, 102–4, 120
ancestor properties, 214–15
AncestorLevel, 214
AncestorType, 214
Anchor, 350
animation, 303, 316, 323–24
  case scenarios, 340–41
  coding, 335–36
  dependency properties, 373
  key frames, 323, 333–35
  lab, animation of controls, 336–37
  non-double types, 332–34
  playback timelines, 330–35
  properties, 324–25
  Triggers, 327–30
Animation, 323–24, 326, 330–35, 363
Application Folder, 446
Application Manifest, 461–63
Application objects, 66–67
Application property, 87
application tasks. *See* commands
Application.Find, 74
Application.GetResourceStream, 189
Application.Resources, 393, 396
Application.Startup, 431
Application.xaml, 393
ApplicationCommands, 73
ApplicationCommands.Print, 419
ApplicationDeployment, 456
ApplicationName, 198
applications, 1
  binary resources, 187
    content files, 190
    embedding, 187, 191–92
    loading, 188–89
    retrieving, 189–91
  communication between, 57
  deploying, 3, 11, 441, 443–44
    Application Manifest, 461–63
    case scenario, 470
    Certificates, 463–64
    ClickOnce, 451–58, 464–65
    Setup projects, 443, 448–49
    Windows Installer, 444–48
    XBAPs, 458–61
  downloading, 3
  events, 66–68
  journal, 24
  localizing, 426–28
    case scenario, 439
    elements, 428–29
    extracting content, 429–30
    lab, practice with, 433–35
    resources, 431
    subdirectories, culture codes, 430–31
    translating content, 430
    UICulture attributes, 428
    validators and converters, 432
  logical resources, 393
  navigating through, 9
  Navigation, 3, 9–10, 444

Page-based, 411
performance, 394
responsiveness, 41
selection of, 14
settings, 86–91
shopping cart, 32
Windows, 3–4, 444
creating, 4–5
displaying, 8–9
lab, creating, 15–16
properties, 5–7
Windows Forms, 3, 5
XBAPs, 3, 11–13, 444
architecture
command, 73
intra-application communication, 57
Argument, 43
Assembly, 379
AssemblyInfo.cs, 379
AssemblyInfo.vb, 379
asynchronous processing, 41–42, 47–48
attached events, 62–63
attached properties, 110, 134, 143
attributes, serializable, 26
audio
lab, creating a media player, 183–85
MediaElement, 179–82
MediaPlayer, 179–82
SoundPlayer, 176–79
Author, 198
AutoReverse, 324, 331

B

Back buttons, 9
Background, 5, 155, 212, 403, 428–29
background processing, 42–43.
See also BackgroundWorker
cancelling, 45–46
changing threads, 47–48
operation cancellation, 42
operation completion, 42
parameters, 43–44
progress reporting, 46
returning values, 44
background, window color, 157–60
BackgroundWorker, 41–42, 45–46, 49–51
BackgroundWorker.ProgressChanged, 46
Balance, 179
BaseOn, 306, 311
BeginAnimation, 335–36
BeginInvoke, 47–48
BeginStoryboard, 328
BeginStoryboardName, 329
BeginTime, 324
binary resources, 187
content files, 190
embedding, 187, 191–92
loading, 188–89
retrieving, 189–91
Binding class, 207, 209–11, 285, 364–65
ADO.NET object binding, 226–27
Binding.Mode, 215–16
case scenarios, 256–57
data
filtering, 246–48
grouping, 243–46
sorting, 241–43
templates, 238–41
elements, binding to, 211–12
hierarchical data, binding, 228–29
labs
data templates and groups, 248–51
database access, 232–35
practice with, 217–18
lists, binding to, 221–26
ObjectDataProvider, 230–31
objects, binding to, 212–15
UpdateSourceTrigger, 216–17
validation rules, 282–83
XmlDataProvider, 231–32
Binding property, 315
Binding.Mode, 215–16
BindingInError, 285
BindingListCollectionView, 225
BindingListCollectionView.CustomFilter, 247–48
bindings, command, 75–78
BitmapDecoder, 199
BitmapDecoder.Create, 199
BitmapFrame, 199
BitmapImage, 199
BitmapImage.Metadata, 199
BitmapMetadata, 198–99
block elements, 401
BlockUIContainer, 409–10, 415
flow documents, 402–4
List, 406
Paragraph, 405
Section, 409
Table, 407–8
BlockUIContainer, 409–10, 415
BlockUIElement, 412–13
Blue channel, 156
Bnzier curve, 334–35
Boolean?, 104
BorderBrush, 5, 155, 404

BorderThickness, 5, 404
Both, 195
Box, 407
Brush, 5, 48–49, 155–56, 196, 403–4
Brush.Freeze, 156
brushes, 155–56
bubbling events, 60, 63, 286
bubbling, commands, 76–77
BufferingProgress, 179
BuildAction, 187
built-in commands, 73
Button, 62–63, 101, 103–5
  Back, 9
  clipping, 171
  databinding, 212
  Forward, 9
  lab, creating control templates, 367–69
  Toolbar, 121
  transforming, 170
  XAML, 101–2
Button.Click event, 359–60
ButtonBase class, 104

C

CameraManufacturer, 198
CameraModel, 198
CancelAsync, 42, 45–46
CancellationPending, 42, 45–46
CanExecute, 77–78
CanExecuteRoutedEventArgs, 77–78
CanGoBack, 24–25
CanGoForward, 24
CanMinimize, 6
CanResize, 6
CanResizeWithGrip, 6
Canvas controls, 101, 142–43
Canvas.ZIndex, 142–43
cart, shopping, 32
CAS (Code Access Security), 460
case scenario
  animation of controls, 340–41
  controls, streaming stock quotes, 151
  custom controls, 386
  data conversion and validation, 301–2
  databinding, 256–57
  deploying applications, 470
  designing user interfaces, 54
  international business, 439
  multimedia, 205
  updating applications, 470
  user input, validating, 95
  user interface, user input, 96
CenterOwner, 7
CenterScreen, 7
Certificates, 463–64
change notification, 282, 287–94, 373, 394
channels, color, 156
characters, 351
Checkbox control, 104, 117, 121
CheckFileExists, 346
CheckForUpdate, 456
CheckForUpdateCompleted, 457
CheckForUpdateCompletedEventArgs, 457
CheckPathExists, 346
child controls, 4, 9, 143–44
Children, 326
Children.Add, 143–44
Children.Remove, 144
Chinese language, 426
Circle, 407
Classic.xaml, 379
Click, 62, 103–5
ClickOnce, 191, 443–44, 451–58, 464–65
Clip, 171
clipping, graphics, 171
Close, 8–9
Code Access Security (CAS), 460
code execution, 41, 59–61, 75
coding
  animations, 335–36
  databinding, 211–13
  styles, 309–10
collections, databinding, 223–26, 246–48
CollectionViewSource, 241
CollectionViewSource.GetDefaultView, 224–25
color, 348–49, 428–29
  control, 108–9
  gradients, 157–61
  themes, 378–80
Color, 48, 158
Color.FromArgb, 157
ColorAnimation, 332
ColumnDefinitions, 133–37
columned page view, 416
columns, flow documents, 416
columns, grid, 133–38
CombinedGeometry, 166–68
ComboBox, 117–18, 121
ComboBox.Content, 118
ComboBox.Text, 118
Command, 73–74, 120
command handler, 73
Command.CanExecute, 77–78

Command.Execute, 75
CommandBinding, 73, 76
commands, 57
  architecture, 73
  configuring, 72–73
    bubbling, 76–77
    custom, 78–80
    disabling, 77
    handlers and bindings, 73, 75–78
    implementing, 73–74
    invoking, 74–75
  hyperlinks, 411
  lab, creating custom, 80–83
  menus, 119–21
Comment, 198
Compare, 242–43
compiling, embedded resources, 187
ComponentCommands, 73
compressed files (.wav), 176–79
Condition, 315
configuring
  application settings, 86–91
  commands, 72–73
    custom, 78–80
    disabling, 77
    handlers and bindings, 73, 75–78
    implementing, 73–74
    invoking, 74–75
  databinding, 207–9
    ADO.NET object binding, 226–27
    Binding.Mode, 215–16
    case scenarios, 256–57
    data templates, 238–41
    data, filtering, 246–48
    data, grouping, 243–46
    data, sorting, 241–43
    elements, binding to, 211–12
    hierarchical data, 228–29
    lab, data templates and groups, 248–51
    lab, database access, 232–35
    lab, practice with, 217–18
    lists, binding to, 221–26
    ObjectDataProvider, 230–31
    objects, binding to, 212–15
    UpdateSourceTrigger, 216–17
    XmlDataProvider, 231–32
  events, 59–61
    application level, 66–68
    EventManager, 63
    handlers, 62–63, 66
    routed events, 61–62, 64–65, 68–69
  lab, change notification and validation, 289–94
  page-based navigation, 21
    event handling, 27–30
    hosting pages in frames, 21
    hyperlinks, 22–23
    journal, using, 25–27
    NavigationService, 23–25
    PageFunction objects, 30–32
    simple, 32
    structured, 32
    using pages, 21
    XBAPs, 11
constructor, 27
ConstructorParameters, 230
containers, flow document, 416
ContainerStyle, 244
ContainerStyleSelector, 244
content
  adding, 153–54
  binary resources, 187
    content files, 190
    embedding, 187, 191–92
    loading resources, 188–89
    retrieving, 189–91
  graphics, 155
    brushes, 155–56
    clipping, 171
    Ellipse, 164–65
    hit testing, 171–72
    ImageBrush, 161–62
    lab, practice with, 172–73
    Line, 165
    LinearGradientBrush, 157–60
    Polygon, 165–68
    Polyline, 165
    RadialGradientBrush, 160–61
    Rectangle, 164–65
    shapes, 163–64
    SolidColorBrush, 156–57
    Transforms, 168–70
    VisualBrush, 163
  images, 194
    bitmap metadata, 198–99
    lab, practice with, 200–1
    stretching and sizing, 194–96
    transforming graphics, 196–98
  managing, 153–54
  multimedia
    case scenarios, 205
    lab, creating a media player, 183–85
    MediaElement, 179–82, 190
    MediaPlayer, 179–82, 190, 196
    media-specific event handling, 182–83
    SoundPlayer, 176–79
Content, 101–2, 211–13, 360

content controls, 101–5, 111, 372
ContentControl, 4, 101–5, 372
ContentPresenter, 360–61
ContentTemplate, 240
ContentType, 189
ContextMenu, 119, 121, 124
Control class, 372
control containment heirarchy, 59–60
control templates, 359
  creating, 367–69, 378–79
  part names, predefined, 366
  source code, 366
  Styles, 365
  templated parent properties, 363–65
  Triggers, 362–63
Control.ContextMenu, 121
Control.Triggers, 313
controls, 101. *See also* content controls; individual control names
  animation of controls, 336–37
  case scenarios
    animation of controls, 340–41
    international business, 439
    streaming stock quotes, 151
  commands, associating with, 74
  control templates, 359–62
    lab, creating, 367–69
    part names, predefined, 366
    source code, 366
    Styles, 365
    templated parent properties, 363–65
    Triggers, 362–63
  custom, 378
    case scenario, 386
    choosing, 373
    consuming controls, 377
    creating, 372, 376–77
    dependency properties, 373–75
    lab, creating custom controls, 380–83
    selecting, 373
    theme-based appearance, 378–80
    user controls, 376
  item controls
    binding to lists, 221–23
    ComboBox, 117–18, 121
    ContextMenu, 119, 121, 124
    lab, practice with, 124–26
    ListBox, 101, 116–17, 121, 124
    menus, 119–21
    StatusBar, 123
    Toolbar, 119, 121–23
    TreeView, 101, 118–19
    virtualization, 123–24
  layout controls, 4, 101, 130–31
    aligning content, 144–45
    Canvas, 101, 142–43
    child elements, accessing, 143–44
    DockPanel, 139–42, 146–48
    Grid, 101, 110, 131–37
    HorizontalAlignment, 131–33, 135, 138
    lab, practice with, 146–48
    Margin, 131–33, 135
    StackPanel, 101, 123–24, 131–32, 138
    UniformGrid, 137–38
    VerticalAlignment, 131–33, 136
    WrapPanel, 139
  menus, 121–22
  Navigation applications, 9
  Page objects, 9
  styles, 309
  tab order, 111
  user, creating, 372
  virtualization, 123–24
  Windows Forms
    dialog boxes, 345–49
    MaskedTextBox, 351–52
    PropertyGrid, 353–54
    WindowsFormsHost, 349–51
  WindowsFormsHost, 349–51
ControlTemplate, 361–63
ControlTemplate.Triggers, 362–63
Convert, 245–46, 261–62, 273–76, 432
ConvertBack, 245–46, 261–62, 265, 273–76, 432
converting data, 261
  bound data, formatting, 273
  case scenario, 301–2
  formatting, conditional, 268–69
  IValueConverter, 261–64
  lab, string and conditional formatting, 276–79
  localizing data, 271, 432
  multi-value converters, 273–76
  objects, return, 268–69
  string formatting, 264–67
Copyright, 198
CreatePrompt, 346
creating
  application settings, 87
  commands, custom, 78–83
  content, 153–54
  control templates, 359–62, 367–69
  data groups, custom, 245–46
  data-based objects, 261
    bound data, formatting, 273
    case scenario, data conversion, 301–2
    formatting, conditional, 268–69

IValueConverter, 261–64
lab, string and conditional formatting, 276–79
localizing data, 271
multi-value converters, 273–76
returning objects, 268–69
string formatting, 264–67
dialog boxes, 8
event handlers, 28, 66–69
graphics, 155
brushes, 155–56, 196
clipping, 171
Ellipse, 164–65
hit testing, 171–72
ImageBrush, 161–62
lab, practice with, 172–73
Line, 165
LinearGradientBrush, 157–60
Polygon, 165–68
Polyline, 165
RadialGradientBrush, 160–61
Rectangle, 164–65
shapes, 163–64
SolidColorBrush, 156–57
Transforms, 168–70
VisualBrush, 163
labs
control templates, 367–69
custom commands, 80–83
custom controls, 380–83
flow documents, 421–22
media player, 183–85
Navigation applications, 16–17
Setup project, 448–49
user interface, 111–12
Windows applications, 15–16
XBAPs, 17–19
Navigation applications, 10, 16–17
resource dictionary, 395–96
Setup projects, Windows Installer, 443, 448–49
styles, 308–10, 318–20
user controls, 372, 376
Windows applications, 4–5, 15–16
XBAPs, 11–12, 17–19
Ctrl key, 74
culture
case scenario, localizing applications, 439
elements, localizable, 428–29
extracting content, 429–30
localizing, 426–28, 433–35
resources, 431
subdirectories, culture codes, 430–31
translating content, 430
UICulture attributes, 428
validators and converters, 432
Culture, 271
CultureInfo, 431–32
currency formats, 265, 352
CurrentDeployment, 456
CurrentItem, 224–26
CurrentPosition, 224
CurrentThread.CurrentUICulture, 431
CurrentUICulture, 426
Cursor, 5
Custom Actions Editor, 448
custom dialog boxes, 8. *See also* dialog boxes
CustomContentState class, 26
CustomSort, 242–43
Cyrillic language, 426

D

data
Boolean, 104
change notification, 287–89
converting, 261
bound data, formatting, 273
case scenario, 301–2
formatting, conditional, 268–69
IValueConverter, 261–64
lab, string and conditional formatting, 276–79
localizing, 271
multi-value converters, 273–76
objects, return, 268–69
string formatting, 264–67
culture settings, 432
databinding, 207–9
ADO.NET object binding, 226–27
Binding.Mode, 215–16
case scenarios, 256–57
data templates, 238–41
data, grouping, 243–46
data, sorting, 241–43
elements, binding to, 211–12
filtering, 246–48
hierarchical data, 228–29
lab, data templates and groups, 248–51
lab, database access, 232–35
lab, practice binding, 217–18
lists, binding to, 221–26
ObjectDataProvider, 230–31
objects, binding to, 212–15
UpdateSourceTrigger, 216–17
XmlDataProvider, 231–32

lab, online pizza ordering, 32–34
settings, 86–87
templates, 248–51
validation, 282
binding rules, 282–83
case scenario, 301–2
custom rules, 283–84
error handling, 284–87
ExceptionValidationRule, 283
lab, configuring, 289–94
ObservableCollection, 288–89
data triggers, 315
databases, 11, 14, 228–29
databinding, 207–9, 364–65
ADO.NET object binding, 226–27
Binding.Mode, 215–16
case scenarios, 256–57
change notification, 287–89
data templates, 238–41
dependency properties, 373
elements, binding to, 211–12
ExceptionValidationRule, 283
filtering data, 246–48
grouping data, 243–46
hierarchical data, 228–29
labs
data templates and groups, 248–51
database access, 232–35
practice with, 217–18
lists, binding to, 221–26
Multibinding, 275–76
ObjectDataProvider, 230–31
objects, binding to, 212–15
sorting data, 241–43
UpdateSourceTrigger, 216–17
validation rules, 282–83
XmlDataProvider, 231–32
DataContext, 213–14, 226–27
DataRelation, 228–29
DataSet, 227
DataTable, 226–27
DataTrigger, 312
DataView, 247–48
date, formats, 267, 352
DateTake, 198
DateTime.ToString, 267
Deactivated events, 67
DecelerationRatio, 324, 331
Decimal, 407
decimal characters, 352, 432
Default Windows XP theme, 379
DefaultLocation, 446
delegates, 48
dependency properties, 373–75, 394
DependencyObject class, 172, 373
deploying applications, 441, 443–44
Application Manifest, 461–63
case scenario, 470
Certificates, 463–64
ClickOnce, 451–58, 464–65
downloading, 3
Setup projects, 443, 448–49
to server, 3
to Web site, 3
Windows Installer, 443–48
XBAPs, 11, 458–61
Designer, Visual Studio, 5
desktop applications, 2, 9, 190–91. *See also* applications
dialog boxes, 14
creating custom, 8
file dialog boxes, 345–47
lab, practice with Windows Forms elements, 354–56
MaskedTextBox, 351–52
PropertyGrid, 353–54
WindowsFormsHost, 349–51
dictionaries, resource, 395–98
digital signatures, 463–64
digits, 351
direct events, 60
disabling commands, 77
Disc, 407
DiscreteKeyFrame, 334–35
Dispatcher, 41, 47–51
DispatcherPriority, 48
DispatcherUnhandledException, 67
DispatcherUnhandledExceptionEventArgs.Handled, 67
DisplayMemberPath, 221–22, 226, 241
DLL (Dynamic Link Library), 377, 418, 430
Dock, 139, 350
DockPanel, 139–42, 146–48
DockPanel.Dock, 139
Document, 231
DocumentPaginator, 420
documents, 401
flow documents, 401
block elements, 405–10
containers, 416
creating, 402–3, 421–22
formatting, 403–4
inline elements, 410–15
scaling text, 417
white space, 415
printing, 418–20
XPS documents, 418

DocumentViewer, 418–19
DoubleAnimationUsingPath, 324
downloading applications, 3, 107
DownloadProgress, 180
DownOnly, 195
DoWork, 41–44
DoWorkEventsArgs, 43–44
drag-and-drop functionality, 14
Drawing, 196–98
DrawingGroup, 196
DrawingImage, 197–98
Duration, 325, 331–32
Dynamic Link Library (DLL), 377, 418, 430
DynamicResource, 378, 393–95, 397

E

e.Action, 286
e.Cancel, 30
e.CanExecute, 77–78
EditingCommands, 73
Element, 213, 217
ElementName, 209–11
elements. *See also* databinding; objects; Resources
  animation, 323–24
  block elements, 401
    BlockUIContainer, 409–10
    flow documents, 401–4
    List, 406
    Paragraph, 405
    Section, 409
    Table, 407–8
  control templates, 359–62
  custom, 373
  databinding, 211–12
  file associations, 461–63
  flipping, 170
  inline elements, 401, 410
    Bold, 410–11
    Figure, 414–15
    Floater, 412–14
    flow documents, 401–4
    Hyperlink, 411
    InlineUIContainer, 415
    Italic, 410–11
    LineBreak, 411–12
    Run, 410
    Span, 412
    Underline, 410–11
  inline flow, 22
  localization, 428–29
  property setters, 306–7
  styles, setting, 308
  transforming, 170
  Underline, 410–11
  user interface (UI), 4, 72
  visual, 163, 420
Ellipse, 164–65
EllipseGeometry, 166, 171
embedded files
  binary resources, 187
  content files, 190
  lab, using embedded resources, 191–92
  loading, 188–89
  retrieving, 189–91
English language, 426
EnterActions, 313
Error objects, 285
ErrorCondition, 283–84
ErrorContent, 285
ErrorException, 183
Esc key, 103
EvenOdd, 166
event handlers, 59–61
  application level, 67–68
  Application.Startup, 431
  attaching, 62–63
  commands, 72, 75–78
  creating, 28, 66–68
  Hyperlink, 411
  lab, routed events, 68–69
  media-specific, 182–83
  setters, 307
  Validation.Error, 285–87
  Windows Forms controls, 350
  XAML, 62
Event property, 307
event triggers, 315–16
EventArgs, 62
EventManager, 63, 66
events, 57
  application level, 66–68
  attached, 62–63
  bubbling, 60, 63, 286
  Button.Click, 359–60
  Click, 103–5
  configuring, 59–61
    application-level events, 66–68
    EventManager, 63
    handlers, 62–63, 66
    routed events, 64–65, 68–69
    RoutedEventArgs, 61–62
  defining, 64–65
  direct, 60
  DoWork, 41–44

EventManager, 63
lab, routed practice, 68–69
navigation, handling, 27–30
PageFunction, 30–32
raising, 65
registration, 63
ReturnEventArgs, 30–31
routed, 57, 61–62, 64–65, 68–69, 76–77, 183
setters, 307
tunneling, 60–61, 63
ValueChanged, 110
EventSetter, 307
EventTrigger, 313, 327
Exception, 285
ExceptionRoutedEventArgs, 183
exceptions, 48, 67, 183, 283
ExceptionValidationRule, 283
Exclude, 167
ExecutedRoutedEventArgs, 75–77
Exit, 67
ExitActions, 313
Extensible Application Markup Langugage (XAML), 1
attached properties, 110
binary resources, 190–91
Button controls, 101–2
Canvas, 142
ContextMenu, 121
custom commands, 80
event handlers, 62
ListBox controls, 116–17
menus, 119–21
multimedia formats, 179
resources, accessing, 393
TreeView controls, 118–19
extensions, filename, 346

F

FallbackValue, 209
Figure, 414–15
file
associations, 461–63
file system, 11–12, 14, 444
File System Editor, 445–46
FileName, 346
filename extensions, 346
filename filter, 346
FileNames, 346
files
associations, 444
binary resources, 187
content files, 190
embedding, 187
lab, embedded resources, 191–92
loading, 188–89
loose files, retrieving, 189–91
retrieving manually, 189
overwriting, 347
sharing, 443–44
files downloading, 107
Files Of Type dialog box, 346
Files Type Editor, 448
Fill, 155
Polygon, 165
Shape, 164
Stretch, 162, 164, 195
FillBehavior, 325, 331
FillRule, 165, 167
Filter, 247, 346
filtering, 246–48, 346
FindAncestor, 214
FindResource, 396–97
flipping, elements, 170
FlipX, 163
FlipXY, 163
FlipY, 163
Floater, 412–14
flow documents, 401
block elements, 405–10
containers, 416
creating, 402–3, 421–22
formatting, 403–4
inline elements, 410–15
scaling text, 417
white space, 415
FlowDirection, 130, 138–39, 428–29
FlowDocumentPageViewer, 416–17, 419
FlowDocumentReader, 415–17, 419
FlowDocumentScrollContainer, 415
FlowDocumentScrollViewer, 416–17, 419
FlowPanel, 60
FontFamily, 403
FontSize, 403
FontStretch, 403
FontStyle, 403
FontWeight, 404
FontWidth, 428–29
Foreground, 5, 155, 403, 428–29
Forever, 331–32
FormatProvider, 352
formatting
data, 261
bound data, 273
case scenario, data conversion, 301–2
conditional formatting, 268–69

IValueConverter, 261–64
lab, string and conditional formatting, 276–79
localizing, 271
multi-value converters, 273–76
objects, return, 268–69
strings, 264–67
document text, 401–4
flow documents, 403–4
block elements, 405–10
containers, 416
creating, 421–22
inline elements, 410–15
scaling text, 417
white space, 415
XPS documents (XML Paper Standard), 418
Forward buttons, 9
fragment navigation, 23, 28
FragmentNavigation, 28
Frame control, 21, 199
frames, hosting pages in, 21
FrameworkPropertyMetadata, 375
Freezable class, 41, 48–49, 156, 364–65
French language, 427

G

Generic.xaml, 376, 379
Geometry, 196–97
Geometry objects, 166–68, 171
GeometryCombineMode, 167
GeometryDrawing, 196–97
GeometryGroup, 166–67
gestures, 72, 74
GetContentState, 27
GetNavigationService, 23
GetRoutedEvents, 64
GetRoutedEventsForOwner, 64
GetValue, 375
GlyphRun, 196
GlyphRunDrawings, 196
GradientStop, 158, 160
graphic handles, 108–9
graphics. *See also* visual effects
brushes, 155–56
clipping, 171
creating, 155
Ellipse, 164–65
freezable objects, 48–49
hit testing, 171–72
ImageBrush, 161–62
lab, practice with, 172–73
Line, 165
LinearGradientBrush, 157–60
managing, 153–54
Polygon, 165–68
Polyline, 165
RadialGradientBrush, 160–61
Rectangle, 164–65
shapes, 163–64
SolidColorBrush, 156–57
transforming to images, 196–98
Transforms, 168–70
VisualBrush, 163
Green channel, 156
Grid, 9, 132–37
attached properties, 110
block elements, 409–10
buttons, 62–63
databinding, 213
margins, 131–32
Grid.Column, 134
Grid.ColumnSpan, 134, 137
Grid.Row, 134
Grid.RowSpan, 135
GridSplitter, 134–37, 146–48
GroupHeader, 244
grouping, data, 243–46
GroupName property, 104
GroupStyle, 244

H

Handled, 61, 63
Handler property, 307
handlers
command, 73, 75–78
event, 59–61
application level, 67–68
Application.Startup, 431
attaching, 62–63
commands, 72, 75–78
creating, 28, 66–68
Hyperlink, 411
media-specific, 182–83
routed events, 68–69
setters, 307
Validation.Error, 285–87
Windows Forms controls, 350
XAML, 62
HasAudio, 180
HasVideo, 180
HeaderTemplate, 244
HeaderTemplateSelector, 244
Height, 5–6, 130
Grid, 133–37
Shape, 164

hexadecimal notation, 156
hierarchical data, 228–29
history, 3, 9. *See also* journals
hit testing, 171–72
HitTestResult, 171–72
HitTestResult.VisualHit, 172
HorizontalAlignment, 131–33, 135, 138, 415
HorizontalAnchor, 414–15
HorizontalContentAlignment, 131
HorizontalOffset, 414
HorizontalScrollBar, 107
HTML (Hypertext Markup Language), 11, 22, 407
hyperlinks, 22–23, 411
Hypertext Markup Language (HTML), 11, 22, 407

I

IBindingList, 225
ICollectionView, 246–48
ICollectionView.GroupDescriptions, 243
ICollectionViews, 224–26, 241–43
ICommandSource, 74
IComparer, 242–43
Icon, 6
icons, 447–48
IDE (integrated development environment), 4–5
IDocumentPaginatorSource, 420
IEnumerable, 225
IList, 225
Image, 106, 188–89, 194, 199, 413
Image.Stretch property, 106
ImageBrush, 161–62
ImageDrawing, 196
images
  binary resources, 187
    content files, 190
    embedding, 187, 191–92
    loading, 188–89
    loose files, retrieving, 189–91
  bitmap metadata, 198–99
  case scenarios, 205
  display of, 106
  Image element, 194
  ImageBrush, 161–62
  lab, practice with, 200–1
  retrieving manually, 189
  stretching and sizing, 194–96
  transforming graphics, 196–98
ImageSource, 6, 161, 194, 199, 428–29
IMultiValueConverter, 273–76
index, 116, 143–44
individual controls, 101
information passing, navigation events, 29
inheritance, styles, 311–12, 317
InitialDirectory, 346
inline elements, 401, 410
  Bold, 410–11
  Figure, 414–15
  Floater, 412–14
  flow documents, 402–4
  Hyperlink, 411
  InlineUIContainer, 415
  Italic, 410–11
  LineBreak, 411–12
  Run, 410
  Span, 412
  Underline, 410–11
inline flow elements, 22
INotifyPropertyChanged, 287
input gestures, 72, 74
InputGestures, 74
Installation Folder URL, 453
integrated development environment (IDE), 4–5
IntelliSense, 88
international business, 426–28
  case scenario, 439
  elements, localizable, 428–29
  extracting content, 429–30
  lab, localizing an application, 433–35
  resources, 431
  subdirectories, culture codes, 430–31
  translating content, 430
  UICulture attributes, 428
  validators and converters, 432
Internet applications, 443–44
Internet Explorer, 11
Internet security zone, 14
Intersect, 167
Invoke, 47–48
IProvideCustomContentState interface, 26–27
IsAsynchronous, 230
IsBusy, 42
IsCancel, 103
IsChecked, 104
IsCurrentAfterLast, 224
IsCurrentBeforeFirst, 224
IsDefault, 103
IsDropDownOpen, 118
IsEditable, 118
IsEnabled, 6
IsLoadCompleted, 176
IsMainMenu property, 119
isolated storage, XBAPs, 12–13
IsolatedStorageFileStream class, 13
IsolateStorageFile class, 12
IsReadOnly, 107, 118

IsSynchronizedWithCurrentItem, 222, 225–26, 228
IsToolBarVisible, 417
IsValid, 283–84
item controls, 101
  binding to lists, 221–23
  ComboBox, 117–18, 121
  ContextMenu, 119, 121, 124
  lab, practice with, 124–26
  ListBox, 101, 116–17, 121, 124
  menus, 119–21
  StatusBar, 123
  Toolbar, 119, 121–23
  TreeView, 101, 118–19
  virtualization, 123–24
ItemsPresenter, 361
ItemsSource, 222, 226
ItemsTemplate, 240
ItemTemplate, 222, 241
IValueConverter, 245–46, 261, 432

J

JournalEntry, 25
JournalEntryName property, 26
journals, 3, 25
  adding items, 26–27
  lab, online pizza ordering, 34–38
  Navigation applications, 9
  NavigationService, 24
  removing items, 25, 32

K

Key, 241, 396
key frame-based animations, 323, 333–35
keyboard gestures, 74
keyboard shortcuts, 102–4, 120
KeyboardNavigation.IsTabStop, 111
KeySpline, 334
KeyTime, 333
Keywords, 199

L

Label controls, 60, 101–3, 123
  databinding, 211–13, 226
labs
  animation of controls, 336–37
  BackgroundWorker, 49–51
  change notification and validation, configuring, 289–94
  commands, creating custom, 80–83
  control templates, creating, 367–69
  controls, creating custom, 380–83
  data templates and groups, 248–51
  databases, accessing, 232–35
  databinding, practice with, 217–18
  embedded resources, using, 191–92
  flow documents, creating, 421–22
  graphics, practice with, 172–73
  images, practice with, 200–1
  item controls, practice, 124–26
  layout controls, practice, 146–48
  localizing an application, 433–35
  media player, creating, 183–85
  Navigation applications, creating, 16–17
  online pizza ordering, 32–38
  publishing with ClickOnce, 464–65
  resources, practice with, 397–98
  routed events, practice, 68–69
  settings, practice with, 89–91
  string and conditional formatting, 276–79
  styles, creating high-contrast, 318–20
  user interface, building, 111–12
  Windows applications, creating, 15–16
  Windows Forms elements, practice with, 354–56
  XBAP, creating, 17–19
languages, 426
LastChildFill, 139, 141–42
Launch Conditions Editor, 448
layout controls, 4, 101, 130–31
  aligning content, 144–45
  Canvas, 101, 142–43
  case scenario, streaming stock quotes, 151
  child elements, accessing, 143–44
  control templates, 360
  DockPanel, 139–42, 146–48
  Grid, 101, 110, 131–37
  HorizontalAlignment, 131–33, 135, 138
  lab, practice with, 146–48
  Margin, 131–33, 135
  Navigation applications, 9
  StackPanel, 101, 123–24, 131–32, 138
  UniformGrid, 137–38
  user interface, 4
  VerticalAlignment, 131–33, 136
  WindowsFormsHost, 350
  WrapPanel, 139
Left, 6, 350
letters, 351
Line, 165
line breaks, 415
linear animations, 323
LinearGradientBrush, 157–60
LinearGradientBrush.EndPoint, 158
LinearGradientBrush.Spread, 160

LinearGradientBrush.StartPoint, 158
LinearKeyFrame, 334–35
LineGeometry, 166
LineHeight, 404
LineStackingStrategy, 404
links, page-based navigation, 22–23
List, 412
list-based controls. *See* item controls
ListBox controls, 101, 116–17, 409–10
  ContextMenu, 121
  databinding, 221–23, 226, 238–40
  virtualizing, 124
ListBox.SelectedIndex, 116
ListBox.SelectedItem, 116
ListBoxItem, 116–17
ListCollectionView, 225, 242–43
ListCollectionView.CustomSort, 242–43
ListItem, 406
lists, databinding to, 221–26
ListView, 124
Load, 176
LoadAsync, 177
LoadComplete, 177
LoadCompleted, 28
LoadedBehavior, 180
LoadTimeout, 177
localizing applications, 426–28
  case scenario, 439
  elements, 428–29
  extracting content, 429–30
  lab, practice with, 433–35
  resources, 431
  subdirectories, culture codes, 430–31
  translating content, 430
  UICulture attributes, 428
  validators and converters, 432
Location, 199
LocBaml, 429–31
logical resources, 212, 389–92
  accessing in XAML, 393
  application resources, 393
  declaring, 392–93
  lab, practice with, 397–98
  resource dictionary, 395–96
  retrieving in code, 396–97
  static and dynamic, 393–95
LostFocus, 217
lowercase characters, 352
LowerLatin, 407
LowerRoman, 407
Luna.Metallic.xaml, 379
Luna.NormalColor.xaml, 379

M

managing
  application responsiveness, 41
  binary resources, 187
    content files, 190
    embedding, 187, 191–92
    loading, 188–89
    retrieving, 189–91
  content, 153–54
  images, 194
    bitmap metadata, 198–99
    case scenarios, 205
    Image element, 194
    ImageBrush, 161–62
    lab, practice with, 200–1
    retrieving, 189
    stretching and sizing, 194–96
    transforming graphics, 196–98
Margin, 131–33, 135, 164, 404
MarkerStyle, 406
Mask, 351
MaskedTextBox, 351–52, 354–56, 386
MaskedTextProvider, 351
MatrixAnimationUsingPath, 324
MatrixTransform, 169
MaxHeight, 131
MaxWidth, 131
Media Player 11, 179
MediaCommands, 73
MediaElement, 179–83, 190
MediaEnded, 182
MediaFailed, 182–83
MediaOpened, 182–83
MediaPlayer, 179–83, 190, 196
memory, NavigationService, 24
Menu, 101, 124
MenuItem, 119–21
menus, 72, 119–22
metadata, 317
Metadata, 199
MethodName, 230
MethodParameters, 230
Microsoft Windows Installer, 190–91, 443–48
Microsoft Windows Media Player, 179
Microsoft Windows Vista, 7, 179
Microsoft Windows XP, 179
migrating, settings, 458
MinHeight, 131
MinWidth, 131
mnemonic keys, 102–3
Mode, 210
mouse gestures, 74
MouseDown, 60

MouseLeave, 60
MoveCurrentTo, 224
MoveCurrentToFirst, 224
MoveCurrentToLast, 224
MoveCurrentToNext, 224–25
MoveCurrentToPosition, 224
MoveCurrentToPrevious, 224–25
Msbuild.exe, 429
Multibinding, 275–76
MultiDataTrigger, 313, 315
multimedia
  case scenarios, 205
  lab, creating a media player, 183–85
  MediaPlayer, 179–82
  media-specific event handling, 182–83
  SoundPlayer, 176–79
Multiselect, 346
multithreaded code, 48–49
MultiTrigger, 312
multi-triggers, 314
My, 88
myFilter, 247

N

Name, 86–87, 363
NaturalDuration, 180
NaturalVideoHeight, 180
NaturalVideoWidth, 180
Navigate, 23–25
Navigated, 27
NavigateUri, 22–23
Navigating, 27
navigation, 3, 9
  collections and lists, 223–26
  flow documents, 416
  fragment, 23, 28
  journal, using, 25–27
  NavigationService, 23–25
  page-based, 21
    event handling, 27–30
    hosting pages in frames, 21
    hyperlinks, 22–23
    PageFunction objects, 30–32
    simple, 32
    structured, 32
    using pages, 21
  XBAPs, 11
Navigation applications, 3, 9
  creating, 10
  deploying, 444, 451
  lab, creating, 16–17
NavigationCommands, 73
NavigationFailed, 28
NavigationProgress, 28
NavigationService, 23–30
NavigationService.AddBackEntry, 27
NavigationService.GoForward, 24
NavigationService.Navigate, 27
NavigationService.Refresh, 24
NavigationService.StopLoading, 24
NavigationStopped, 28
NavigationWindow, 9
NeutralResourcesLanguage, 427
NoBorder, 7
non-double animation, 332–34
nonlinear animation, 333–35
NonZero, 166
NoResize, 6
notification, change, 282, 287–94, 394
NotifyOnSourceUpdated, 210
NotifyOnTargetUpdated, 210
NotifyOnValidationError, 285
null, 75

O

Object array, 273–76
ObjectDataProvider, 230–31
ObjectInstance, 230
objects. *See also* databinding; elements; Resources
  ADO.NET, 247–48
  change notification, 282
  CommandBinding, 76–78
  commands, 72–75
  data-based, 261
    bound data, formatting, 273
    case scenario, data conversion and validation, 301–2
    change notification, 287–89
    formatting, conditional, 268–69
    IValueConverter, 261–64
    lab, string and conditional formatting, 276–79
    localizing data, 271
    multi-value converters, 273–76
    returning objects, 268–69
    string formattting, 264–67
  databinding, 212–15
  displaying, 123
  freezable, 48–49, 156, 364–65
  Geometry, 166–68
  Page, 9–11, 30–32
  PageFunction, 22, 30–32
  read-only, 48–49
  read-write, 48–49
  static, 73, 212

validating, 282
binding rules, 282–83
custom rules, 283–84
error handling, 284–87
ExceptionValidationRule, 283
lab, configuring, 289–94
ObjectType, 230
ObservableCollection, 242–43, 287
Offset, 158
OneTime, 215
OneWay, 215–16
OneWayToSource, 216
online ordering, 32–38
OnReturn, 30–31
opacity, 156
OpacityMask, 156
Open, 181
OpenFile, 347
OpenFileDialog, 345–47
Orientation, 138
OriginalSource, 61
Overflow menu, 121–22
overloads, 29
OverwritePrompt, 346

P

Pack URIs, 188–91
Pad, 160
Padding, 131, 404
Page objects, 9–11, 30–32
page view, 416
page-based navigation, 21
event handling, 27–30
hosting pages in frames, 21
hyperlinks, 22–23, 411
journal, using, 25–27
NavigationService, 23–25
PageFunction objects, 30–32
simple, 32
structured, 32
using pages, 21
XBAPs, 11
PageFunction, 22, 30–38
PageFunction.Returned, 31
pages, 21, 24, 416
Panel, 244
Paragraph, 402, 405–7, 412–14
parent properties, templated, 363–65
Part, 366
part names, 366
Path, 166–68, 364–65
animations, 323–24
databinding, 210–11, 227–28
PathDate, 167
PathGeometry, 166, 168
Pause, 181
PauseStoryboard, 328
Pen, 196
performance
application, 394
freezable objects, 49
permissions, XBAPs, 11
Play, 177, 181
playback timelines, animation, 330–35
PlayLooping, 177
PlaySync, 177
PointAnimationUsingPath, 324
Points, 165
policies, security, 13–14
Polygon, 165–68
Polyline, 165
pop-up menus, 119
Position, 180
positioning, user interface items. *See* layout controls
precedence, properties, 316–18
Predicate, 246–47
PreviewKeyDown, 61
PreviewMouseDown, 60–61
PreviousData, 214
Print, 420
PrintDialog, 419–20
PrintDialog.PrintVisual, 420
PrintDocument, 419–20
printing, 401, 418–20
PrintVisual, 419
priority, delegate execution, 48
processing
asynchronous, 41–42, 47–48
background, 42–43. *See also* BackgroundWorker
cancelling, 45–46
changing threads, 47–48
parameters, 43–44
progress reporting, 46
returning values, 44
progress monitoring, 46, 180, 386
ProgressBar controls, 107–8, 123, 386
ProgressChanged, 42
properties
ancestor, 214–15
attached, 110, 134, 143
databinding, 207–9
ADO.NET object binding, 226–27
Binding.Mode, 215–16
case scenarios, 256–57
data templates, 238–41
elements, binding to, 211–12

filtering data, 246–48
grouping data, 243–46
hierarchical data, 228–29
lab, data templates and groups, 248–51
lab, database access, 232–35
lab, practice binding, 217–18
lists, binding to, 221–26
ObjectDataProvider, 230–31
objects, binding to, 212–15
sorting data, 241–43
UpdateSourceTrigger, 216–17
XmlDataProvider, 231–32
dependency, 373–75, 394
e.Cancel, 30
JournalEntryName, 26
precedence, 316–18
setters, 306–7
triggers, 312, 330
Visual Studio, 5
Windows applications, 5–7
Properties.Settings.Default, 88
Property, 313
property setters, 306–7
property triggers, 312
property value inheritance, 373
property value providers, 373
PropertyChanged, 217, 287
PropertyChangedEventArgs, 287
PropertyGrid, 353–54
PropertyGroupDescription, 243, 245–46
PropertySort, 354
Publish, 452–55
Publishing Folder Location, 452–54
Publishing Options, 454

R

RadialGradientBrush, 160–61
RadialGradientBrush.GradientOrigin, 160
RadioButton controls, 104–5, 121
RadiusX, 160, 164–65
RadiusY, 160, 164–65
RaiseEvent, 65
Rating, 199
ReachFramework, 418
read-only objects, 48–49
read-write objects, 48–49
Rectangle, 164–65
RectangleGeometry, 167
Red channel, 156
Reflect, 160
refresh, 24
RegisterClassHandler, 64
RegisterRoutedEvent, 64
registration, events, 63
registry, 11, 14, 444
Registry Editor, 448
relational data, 233–35
RelativeSource, 210, 213–15, 217, 364–65
RelativeSource.Mode, 214–15
RemoveAt, 144
RemoveBackEntry, 25
RemoveFromJournal, 32
RenderTransform, 168–70
RenderTransformOrigin, 168–70
Repeat, 160
RepeatBehavior, 325, 331–32
Replay, 26–27
ReportProgress, 42, 46
reports, 46
ResizeBehavior, 135, 137
ResizeDirection, 135
ResizeMode, 6
resizing. *See also* Stretch
Canvas containers, 142
Grid, 133–37
images, 106
ToolBar controls, 122–23
windows, 6
resource dictionaries, 395–98
ResourceDictionaryLocation.ExternalAssembly, 380
ResourceDictionaryLocation.None, 379
ResourceDictionaryLocation.SourceAssembly, 379
resources, 378
application, 393, 396
binary, 187
content files, 190
embedding, 187, 191–92
loading, 188–89
retrieving, 189–91
ContextMenu, 121
culture, loading, 431
images, 194
bitmap metadata, 198–99
case scenarios, 205
Image element, 194
ImageBrush, 161–62
lab, practice with, 200–1
retrieving, 189
stretching and sizing, 194–96
transforming graphics, 196–98
logical, 212
satellite assemblies, 431
static and dynamic, 397
templates as, 361

Resources, 306, 308–9, 378
lab, practice with, 397–98
logical, 389–92
accessing in XAML, 393
application resources, 393
declaring, 392–93
resource dictionary, 395–96
retrieving in code, 396–97
static and dynamic, 393–95
responsiveness, application, 41
Result, 44
ResumeStoryboard, 328
return types, PageFunction, 30–31
return value, PageFunction, 31
Returned, 30–31
ReturnEventArgs, 30–31
returning data, online pizza ordering, 32–34
RotateTransform, 169
routed events, 57, 59–62, 183
command bubbling, 76–77
defining, 64–65
lab, practice with, 68–69
registration, 63
RoutedEvent, 61
RoutedEventArgs, 61–63, 183
RowDefinitions, 133–37
rows, grid, 133–38
RuleInError, 285
RunWorkerAsync, 41–44
RunWorkerCompleted, 42, 44
RunWorkerCompletedEventArgs, 44

S

satellite assemblies, 431
Save As dialog box, 346
Save As File Type dialog box, 346
SaveFileDialog, 345–47, 354–56
SaveFileDialog.OpenFile, 347
ScaleTransform, 169–70
ScaleX, 170
ScaleY, 170
scaling text, 417
Scope, 86–87
scroll bars, 107, 116
scroll view, 416
Section, 409
security, 13–14
Certificates, 463–64
code access, 458
Navigation applications, 9
trust environments, 444, 451, 458–61
XBAPs, 11
SeekStoryboard, 328
SelectedIndex, 117
SelectedItem, 117, 226
SelectedObject, 354
SelectionMode, 117
Self, 214
Separator, 121
Serializable attribute, 26
servers, application deployment, 3, 11
SessionEnding, 67
SetBinding, 211–12
SetStoryboardSpeedRatio, 328–29
Setters, 306–7, 313–14, 363
settings, 57
application, 86–91
Internet security, 14
migrating, 458
Settings Editor, 87
Settings object, 88
Setup projects, 443, 445–46, 448–49
SetValue, 375
Shape class, 163–64
shapes, 163–64
clipping, 171
Ellipse, 164–65
Line, 165
Polygon, 165–68
Polyline, 165
Rectangle, 164–65
Transforms, 168–70
shopping cart, 32
shortcut keys, 102–4, 120, 447–48
Show display method, 8–9
ShowDialog display method, 8–9, 347, 420
ShowInTaskbar, 6
ShowsPreview, 135
Silver Window XP theme, 379
simple navigation, 32
SingleBorderWindow, 7
siteOfOrigin, 189–91
SizeToContent, 6
SkewTransform, 169
SkipStoryboardToFill, 328
slider controls, 108–10, 211–12, 416–17
Snaplines, 144–45
snapshots, journal entries, 26
SolidColorBrush, 48, 156–57
Solution Explorer, Visual Studio, 5
SortDescriptions, 241–42
SoundLocation, 177
SoundLocationChanged, 177
SoundPlayerAction, 178–79, 315–16
Source, 210, 212–15, 217, 231

source properties, 208–9
ADO.NET object binding, 226–27
Binding.Mode, 215–16
case scenarios, databinding, 256–57
data templates, 238–41
data, filtering, 246–48
data, grouping, 243–46
data, sorting, 241–43
elements, binding to, 211–12
hierarchical data, binding, 228–29
lab, data templates and groups, 248–51
lab, database access, 232–35
lab, practice binding, 217–18
lists, binding to, 221–26
ObjectDataProvider, 230–31
objects, binding to, 212–15
UpdateSourceTrigger, 216–17
XmlDataProvider, 231–32
Source property, 21, 61, 106, 180, 194
spaces, 415
Span, 412
SpeedRatio, 180, 325, 329, 332
SplineKeyFrame, 334–35
StackPanel controls, 101, 123–24, 131–32, 138
stand-alone windows, 14. *See also* dialog boxes
StartingIndex, 407
StartUp, 67
static objects, 73, 212
StaticResource, 378, 393–95, 397
StatusBar, 123
Stop, 177, 181
StopStoryboard, 328
storage, isolated, 12–13
Storyboard, 324–27, 330–36
Stream, 177, 189
StreamChanged, 177
StreamGeometry, 167
StreamReader, 12–13
StreamWriter, 12–13
Stretch, 132, 135–36
ImageBrush, 162
images, 194–96
Shape, 164
VisualBrush, 163
StretchDirection, 194–96
strings, formatting, 264–67, 276–77
Stroke, 156, 164
StrokeThickness, 164
structured navigation, 32
Style, 305–6, 308, 312, 316, 404
animation triggers, 327
case scenario, custom controls, 386
control templates, 365
logical resources, 392
Style.Triggers, 313
styles, 303, 305
case scenarios, 340–41, 386
creating, 308–10
inheritance, 311–12
lab, creating high-contrast styles, 318–20
properties of, 305–6
property value precedence, 316–18
setters, 306–7
triggers, 312–16
Subject, 199
system culture, 426
System.Globalization.CultureInfo, 431–32
System.Globalization.Info, 426
System.IO.Stream, 347
System.Threading.Thread.CurrentThread.CurrentUICulture, 426
System.Windows.Forms, 348
System.Windows.Forms.Integration, 349
System.Windows.Media.Animation, 323
System.Windows.Media.Color, 348
System.Windows.Media.ImageSource class, 106
System.Windows.Resources.StreamResourceInfo, 189
System.Windows.Xps.Packaging, 418
SystemColors, 378
SystemColors.WindowColor, 212
SystemDeployment, 456
SystemFonts, 378
SystemParameters, 378

T

Tab key, 111
tab order, controls, 111
TabIndex, 111
Table, 412–14
TableCell, 407
TableRowGroup, 407
tabs, 415
target properties, 208–9
ADO.NET object binding, 226–27
Binding.Mode, 215–16
case scenarios, databinding, 256–57
data templates, 238–41
data, filtering, 246–48
data, grouping, 243–46
data, sorting, 241–43
elements, binding to, 211–12
hierarchical data, binding, 228–29
lab, data templates and groups, 248–51
lab, database access, 232–35

lab, practice binding, 217–18
lists, binding to, 221–26
ObjectDataProvider, 230–31
objects, binding to, 212–15
UpdateSourceTrigger, 216–17
XmlDataProvider, 231–32
Target property, 103
TargetType, 306, 309, 392
task execution, 41
tasks. *See* commands
Template, 316, 386
TemplateBinding, 364
TemplatedParent, 214, 316, 364–65
templates, 373
control
creating, 378–79
lab, creating, 367–69
parent properties, 363–65
part names, predefined, 366
source code, 366
Styles, 365–66
Triggers, 362–63
controls, custom, 376
resources as, 361
theme-specific, 379–80
templates, data, 238–41, 248–51, 268–69
text
ComboBox, 118
display of, 107
flow documents, 401
block elements, 405–10
containers, 416
creating, 402–3
formatting, 403–4
inline elements, 410–15
lab, creating, 421–22
scaling text, 417
white space, 415
local culture, 428–29
wrapping, 107, 130, 139, 407
XPS documents (XML Paper Standard), 418
TextAlignment, 404
TextBlock, 105–6
TextBox, 101, 107, 121, 217, 351–52
TextWrapping, 107
ThemeInfoAttribute, 379–80
themes, 376, 378–80
Thickness, 131–32
thousands separator, 352
threads, 41, 47–51
ThreeDBorderWindow, 7
thumb, 108–9
Tile, 163
TileMode, 162–63
time formats, 352, 432
time separator, 352
time stamps, 29
Timeline, 326
Title, 7, 199
Toolbar control, 119, 121–23
ToolBar.OverflowMode, 121–22
ToolBarTray, 122–23
Toolbox, 5
ToolTip, 404
ToolWindow, 7
Top, 7, 350
Topmost, 7
ToString, 102, 247, 265
Transformations, 168–70
TransformGroup, 169
transforming
elements, 170
graphics, 196–98
Transforms, 168–70
TranslateTransform, 169
TreeView, 101, 118–19
TreeView.SelectedItem, 119
TreeViewItem, 118–19
triggers, 312–16
Triggers, 306, 308, 312–16, 327–30, 362–63
trust environments, 3, 190–91, 444, 451, 458–61
TryFindResource, 396
tunneling events, 60–61, 63
TwoWay, 216
Type, 86–87
Type array, 273–76

U

UI (user interface). *See* user interface (UI)
UI (user interface) thread, 41
UICulture, 428, 431
Uid, 429
UIElement, 102
UNC (Universal Naming Convention), 191
uncompressed files, .wav, 176–79
underscore (_) symbol, 102–4, 120
Uniform, 162, 164, 195–96
Uniform Resource Identifier (URI), 23–25, 106, 188–91, 194
Uniform Resource Locator (URL), 191
UniformGrid, 137–38
UniformToFill, 162, 164, 195–96
Union, 168
Universal Naming Convention (UNC), 191

UnmanagedMemoryStream, 189
Update, 456
UpdateAsynch, 457
updates, 27, 443–58, 470
UpdateSourceTrigger, 216–17
updateuid, 429
UpOnly, 195
uppercase characters, 352
UpperLatin, 407
UpperRoman, 407
URI (Uniform Resource Identifier), 23–25, 106, 188–91, 194
URL (Uniform Resource Locator), 191
user controls, 378
  creating, 372, 376
    case scenario, custom controls, 386
    choosing, 373
    theme-based appearance, 378–80
  custom, 373–75, 377
  lab, creating custom controls, 380–83
user experience. *See also* user interface (UI)
  Internet, XBAP, 3
  navigation, 3
  ProgressBar controls, 107–8
  settings, saving, 88
user input. *See also* user interface (UI)
  case scenario, user interface, 96
  case scenario, validating, 95
  commands, 72
  gestures, 74
  PageFunction, 32
  window display methods, 8
user interface (UI), 99. *See also* databinding; graphics; multimedia content; visual effects
  attached properties, 110
  case scenario
    designing, 54
    streaming stock quotes, 151
    user input, 96
  control templates, 359
    lab, creating, 367–69
    part names, predefined, 366
    source code, 366
    Styles, 365
    templated parent properties, 363–65
    Triggers, 362–63
  controls, 9, 59, 72, 101–5
  controls, customizing, 343, 372
    case scenario, custom controls, 386
    consuming controls, 377
    creating, 376–77
    dependency properties, 373–75
    lab, creating custom controls, 380–83
  selecting controls, 373
  theme-based appearance, 378–80
  user controls, 372, 376
data display
  data templates, 238–41
  filtering data, 246–48
  grouping data, 243–46
  sorting, 241–43
deploying, 444
elements, 72
Image controls, 106
item controls
  ComboBox, 117–18, 121
  ContextMenu, 119, 121, 124
  lab, practice with, 124–26
  ListBox control, 101, 116–17, 121, 124
  menus, 119–21
  StatusBar, 123
  Toolbar, 119, 121–23
  TreeView, 101, 118–19
  virtualization, 123–24
lab, building, 111–12
lab, updating, 49–51
layout controls, 130–31
  aligning content, 144–45
  Canvas, 101, 142–43
  child elements, accessing, 143–44
  DockPanel, 139–42, 146–48
  Grid, 101, 110, 131–37
  HorizontalAlignment, 131–33, 135, 138
  lab, practice with, 146–48
  Margin property, 131–33, 135
  StackPanel, 101, 123–24, 131–32, 138
  UniformGrid, 137–38
  VerticalAlignment, 131–33, 136
  WrapPanel, 139
localizing (culture variations), 426–28
  case scenario, 439
  elements, 428–29
  extracting content, 429–30
  lab, practice with, 433–35
  resources, 431
  UICulture attributes, 428
  validators and converters, 432
logical resources, 389–92
  accessing in XAML, 393
  application resources, 393
  declaring, 392–93
  lab, practice with, 397–98
  resource dictionary, 395–96
  retrieving in code, 396–97
  static and dynamic, 393–95
Navigation application, 9

ProgressBar controls, 107–8
responsiveness, 41
Slider control, 108–10
subdirectories, culture codes, 430–31
tab order, controls, 111
TextBlock, 105–6
TextBox, 107
translating content, 430
updating, 47–48
Windows applications, 4
Windows Forms controls, 344–45
file dialog boxes, 345–47
lab, practice with, 354–56
MaskedTextBox, 351–52
PropertyGrid, 353–54
Windows properties, 5–7
WindowsFormsHost, 349–51
XBAPs, 11
User Interface Editor, 448
User property, 87
UserControl, 372
user-defined styles, 317
User's Desktop, 446
User's Program Menu, 446

V

Validate, 283–84, 432
ValidateNames, 346
validation
data, 282
binding rules, 282–83
case scenario, 301–2
change notification, 287–89
custom rules, 283–84
error handling, 284–87
ExceptionValidationRule, 283
lab, configuring, 289–94
ObservableCollection, 288–89
navigation events, 27
Validation.Error, 285–87
ValidationCollection, 282–83
ValidationErrorEventArgs, 285–87
ValidationResult, 283–84
ValidationRules, 282–84, 286, 432
validators, culture settings, 432
Value, 86–87, 313
Value property, 87
ValueChanged event, 110
ValueConversion, 262
values
background processing, 44
returning, 31
vertical scroll bars, 107, 116
VerticalAlignment, 131–33, 136
VerticalAnchor, 414
VerticalContentAlignment, 131
VerticalOffset, 414
VerticalScrollBarVisibility, 107
video, 179–85, 196, 205
VideoDrawing, 196
ViewBox, 162
Viewport, 162
views, document, 416
virtualization, item controls, 123–24
VirtualizingStackPanel, 123–24
Visual Basic, settings, 88
Visual class, 163
visual effects
animation, 303, 323–24
coding, 335–36
key frames, 333–35
lab, animation of controls, 336–37
non-double types, 332–34
playback timelines, 330–35
properties, 324–25
with Triggers, 327–30
case scenarios, 340–41
control templates, 359
part names, predefined, 366
source code, 366
Style, 365
templated parent properties, 363–65
Triggers, 362–63
controls, custom, 372, 376–77
case scenario, 386
consuming controls, 377
dependency properties, 373–75
lab, creating custom controls, 380–83
selecting, 373
theme-based appearance, 378–80
user controls, 376
local culture, 428–29
styles, 303, 305
creating, 308–10
inheritance, 311–12
lab, creating high-contrast styles, 318–20
properties of, 305–6
property value precedence, 316–18
setters, 306–7
triggers, 312–16
Windows Forms controls, 344–45
ColorDialog box, 348–49
lab, practice with, 354–56
MaskedTextBox, 351–52
PropertyGrid, 353–54
WindowsFormsHost, 349–51
visual elements, printing, 420

Visual Studio
  creating Windows applications, 4–5
  Designer, 5
  settings editor, 87
  Snaplines, 144–45
  window properties, 7
visual tree, 59–60
VisualBrush, 163
VisualTree, 214
VisualTreeHelper.HitTest, 171–72
volume control, 108–9

W

WCF Web services, 14
Web pages
  page-based navigation, 22
  XBAPs and, 11
Web servers, application deployment, 11
Web services, WCF, 14
Web sites, application deployment, 3
white space, 415
Width, 6–7, 131, 133–37, 164
WidthAndHeight, 6
Window, 60
Window class, 4–7
Window.Resources, 121, 212, 268–69, 392, 396
Window-based applications, 411
windows
  background, 157–60
  borders, 5
  display of, 8–9
  resize, 6
  stand-alone, 14
  style, 7
Windows applications, 3–4, 444
  creating, 4–5
  deploying, 451
  displaying, 8–9
  lab, creating, 15–16
  properties, 5–7
Windows Classic theme, 379
Windows Forms applications, 1, 3, 5, 345–47
Windows Forms controls, 344–45
  ColorDialog dialog box, 348–49
  dialog boxes, 345–49
  lab, practice with, 354–56
  MaskedTextBox, 351–52
  PropertyGrid, 353–54
  WindowsFormsHost, 349–51
Windows Installer, 190–91, 443–49
Windows Internet Explorer, XBAPs, 3
Windows Media Player, 10, 179
Windows Vista, 179, 378–80
Windows XP, 179
Windows.Resources, 308–9
WindowsFormsHost, 354
WindowsFormsHost.Child, 351, 354
WindowsFormsIntegration, 349
WindowsStyle, 5
WindowStartupLocation, 7
WindowState, 7
WindowStyle, 7
worker threads, 47–48
WorkerReportsProgress, 42
WorkerSupportsCancellation, 42, 45–46
Wrap, 107
WrapDirection, 415
WrapPanel, 139
WrapWithOverflow, 107
writing, isolated storage, 12–13

X

X:Key, 392–93
XAML (Extensible Application Markup Language), 1
  attached properties, 110
  binary resources, 190–91
  Button controls, 101–2
  Canvas, 142
  ContextMenu, 121
  custom commands, 80
  event handlers, 62
  ListBox controls, 116–17
  menus, 119–21
  multimedia formats, 179
  resources, accessing, 393
  TreeView controls, 118–19
XAML Browser Application (XBAP). *See* XBAPs
XBAPs, 3, 11, 191, 444
  creating, 11–12
  deploying, 451, 458–61
  isolated storage, 12–13
  lab, creating, 17–19
  web pages and, 11
XML Paper Standard (XPS), 418
XML, databinding, 231–32
XmlDataProvider, 231–32
Xor, 168
XPath, 210, 231–32
XPS documents (XML Paper Standard), 418
XpSDocument, 418

Z

zoom, 401, 416–17
Zoom, 417
Z-order, 142–43